Czytanie kodu. Punkt widzenia twórców oprogramowania open source

Autor: Diomidis Spinellis
Tłumaczenie: Bartłomiej Garbacz
ISBN: 83-7361-555-5
Tytuł oryginału: Code Reading The Open Source Perspective
Format: B5, stron: 448

Książka „Czytanie kodu. Punkt widzenia twórców oprogramowania" open source to pierwszy na rynku podręcznik poświęcony czytaniu kodu źródłowego jako osobnej dziedzinie wiedzy, której znajomość jest przydatna każdemu programiście.
Ponad 600 przykładów, w których wykorzystywane są kody oprogramowania open source, przedstawia sposoby identyfikowania dobrego i złego kodu, czytania go, przeszukiwania pod kątem konkretnych funkcji oraz wykorzystywania umiejętności czytania kodu do poprawy jakości kodów źródłowych pisanych samodzielnie.

- Podstawowe konstrukcje sterujące działaniem programu
- Proste i złożone typy danych
- Struktury i unie
- Dynamiczne zarządzanie pamięcią
- Metody analizy projektów informatycznych
- Konwencje pisania i formatowania kodu źródłowego
- Tworzenie i czytanie dokumentacji
- Architektura systemów

Poznaj umiejętność czytania kodu źródłowego i popraw samodzielnie pisany kod

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Przedmowa **7**

Wstęp **11**

Rozdział 1. Wprowadzenie **15**
1.1. Motywy i metody czytania kodu 16
1.1.1. Kod jako literatura 16
1.1.2. Kod jako model 19
1.1.3. Utrzymanie kodu 20
1.1.4. Rozwój 21
1.1.5. Ponowne wykorzystanie 22
1.1.6. Inspekcje 23
1.2. Jak czytać tę książkę? 24
1.2.1. Konwencje typograficzne 24
1.2.2. Diagramy 25
1.2.3. Ćwiczenia 27
1.2.4. Materiał dodatkowy 27
1.2.5. Narzędzia 28
1.2.6. Zarys treści 28
1.2.7. Debata na temat najlepszego języka 29
Dalsza lektura 30

Rozdział 2. Podstawowe konstrukcje programistyczne **33**
2.1. Pełny program 33
2.2. Funkcje i zmienne globalne 39
2.3. Pętle while, instrukcje warunkowe i bloki 42
2.4. Instrukcja switch 45
2.5. Pętle for 47
2.6. Instrukcje break i continue 50
2.7. Wyrażenia znakowe i logiczne 52
2.8. Instrukcja goto 55
2.9. Refaktoryzacja w skrócie 57
2.10. Pętle do i wyrażenia całkowite 63
2.11. Podsumowanie wiadomości o strukturach sterujących 65
Dalsza lektura 71

Rozdział 3. Zaawansowane typy danych języka C **73**
3.1. Wskaźniki 73
3.1.1. Powiązane struktury danych 74
3.1.2. Dynamiczne przydzielanie struktur danych 74

3.1.3. Wywołania przez referencję ... 75
3.1.4. Uzyskiwanie dostępu do elementów danych ... 76
3.1.5. Tablice jako argumenty i wyniki ... 77
3.1.6. Wskaźniki na funkcje ... 78
3.1.7. Wskaźniki jako aliasy ... 81
3.1.8. Wskaźniki a ciągi znaków ... 82
3.1.9. Bezpośredni dostęp do pamięci ... 84
3.2. Struktury ... 85
3.2.1. Grupowanie elementów danych ... 85
3.2.2. Zwracanie wielu elementów danych z funkcji ... 85
3.2.3. Odwzorowanie organizacji danych ... 86
3.2.4. Programowanie obiektowe ... 87
3.3. Unie ... 89
3.3.1. Wydajne wykorzystanie pamięci ... 89
3.3.2. Implementacja polimorfizmu ... 90
3.3.3. Uzyskiwanie dostępu do różnych reprezentacji wewnętrznych ... 91
3.4. Dynamiczne przydzielanie pamięci ... 92
3.4.1. Zarządzanie wolną pamięcią ... 95
3.4.2. Struktury z dynamicznie przydzielanymi tablicami ... 97
3.5. Deklaracje typedef ... 98
Dalsza lektura ... 100

Rozdział 4. Struktury danych języka C ... 101
4.1. Wektory ... 102
4.2. Macierze i tabele ... 106
4.3. Stosy ... 110
4.4. Kolejki ... 112
4.5. Mapy ... 114
4.5.1. Tablice mieszające ... 117
4.6. Zbiory ... 119
4.7. Listy ... 120
4.8. Drzewa ... 127
4.9. Grafy ... 131
4.9.1. Przechowywanie wierzchołków ... 132
4.9.2. Reprezentacje krawędzi ... 133
4.9.3. Przechowywanie krawędzi ... 136
4.9.4. Właściwości grafu ... 137
4.9.5. Struktury ukryte ... 138
4.9.6. Inne reprezentacje ... 138
Dalsza lektura ... 139

Rozdział 5. Zaawansowane techniki sterowania przebiegiem programów ... 141
5.1. Rekurencja ... 141
5.2. Wyjątki ... 147
5.3. Równoległość ... 151
5.3.1. Równoległość sprzętowa i programowa ... 151
5.3.2. Modele sterowania ... 153
5.3.3. Implementacja wątków ... 158
5.4. Sygnały ... 161
5.5. Skoki nielokalne ... 165
5.6. Podstawienie makr ... 167
Dalsza lektura ... 171

Rozdział 6. Metody analizy dużych projektów ... 173
6.1. Techniki projektowe i implementacyjne ... 173
6.2. Organizacja projektu ... 175
6.3. Proces budowy i pliki Makefile ... 183
6.4. Konfiguracja ... 190
6.5. Kontrola wersji ... 194
6.6. Narzędzia związane z projektem ... 201
6.7. Testowanie ... 205
Dalsza lektura ... 212

Rozdział 7. Standardy i konwencje pisania kodu ... 213
7.1. Nazwy plików i ich organizacja wewnętrzna ... 214
7.2. Wcięcia ... 216
7.3. Formatowanie ... 218
7.4. Konwencje nazewnictwa ... 221
7.5. Praktyki programistyczne ... 224
7.6. Standardy związane z procesem rozwojowym ... 226
Dalsza lektura ... 227

Rozdział 8. Dokumentacja ... 229
8.1. Rodzaje dokumentacji ... 229
8.2. Czytanie dokumentacji ... 230
8.3. Problemy dotyczące dokumentacji ... 241
8.4. Dodatkowe źródła dokumentacji ... 242
8.5. Popularne formaty dokumentacji w środowisku open-source ... 246
Dalsza lektura ... 251

Rozdział 9. Architektura ... 253
9.1. Struktury systemów ... 253
9.1.1. Podejście scentralizowanego repozytorium i rozproszone ... 254
9.1.2. Architektura przepływu danych ... 259
9.1.3. Struktury obiektowe ... 261
9.1.4. Architektury warstwowe ... 264
9.1.5. Hierarchie ... 266
9.1.6. Przecinanie ... 267
9.2. Modele sterowania ... 269
9.2.1. Systemy sterowane zdarzeniami ... 269
9.2.2. Menedżer systemowy ... 273
9.2.3. Przejścia stanów ... 274
9.3. Pakietowanie elementów ... 277
9.3.1. Moduły ... 277
9.3.2. Przestrzenie nazw ... 279
9.3.3. Obiekty ... 283
9.3.4. Implementacje ogólne ... 295
9.3.5. Abstrakcyjne typy danych ... 298
9.3.6. Biblioteki ... 299
9.3.7. Procesy i filtry ... 302
9.3.8. Komponenty ... 304
9.3.9. Repozytoria danych ... 306
9.4. Wielokrotne użycie architektury ... 308
9.4.1. Schematy strukturalne ... 308
9.4.2. Generatory kodu ... 309
9.4.3. Wzorce projektowe ... 310
9.4.4. Architektury dziedzinowe ... 312
Dalsza lektura ... 316

Rozdział 10. Narzędzia pomocne w czytaniu kodu 319
10.1. Wyrażenia regularne 320
10.2. Edytor jako przeglądarka kodu 323
10.3. Przeszukiwanie kodu za pomocą narzędzia grep 326
10.4. Znajdowanie różnic między plikami 333
10.5. Własne narzędzia 335
10.6. Kompilator jako narzędzie pomocne w czytaniu kodu 338
10.7. Przeglądarki i upiększacze kodu 342
10.8. Narzędzia używane w czasie uruchomienia 347
10.9. Narzędzia nieprogramowe 351
Dostępność narzędzi oraz dalsza lektura 353

Rozdział 11. Pełny przykład 355
11.1. Przegląd 355
11.2. Plan działania 356
11.3. Wielokrotne użycie kodu 358
11.4. Testowanie i uruchamianie 363
11.5. Dokumentacja 368
11.6. Uwagi 369

Dodatek A Struktura dołączonego kodu 371

Dodatek B Podziękowania dla autorów kodu źródłowego 375

Dodatek C Pliki źródłowe 377

Dodatek D Licencje kodu źródłowego 385
D.1. ACE 385
D.2. Apache 386
D.3. ArgoUML 387
D.4. DemoGL 388
D.5. hsqldb 388
D.6. NetBSD 389
D.7. OpenCL 390
D.8. Perl 390
D.9. qtchat 393
D.10. socket 393
D.11. vcf 393
D.12. X Window System 394

Dodatek E Porady dotyczące czytania kodu 395

Bibliografia 413

Skorowidz 427

Rozdział 3.

Zaawansowane typy danych języka C

Wskaźniki, struktury, unie, pamięć przydzielana dynamicznie oraz deklaracje nazw typów to podstawowe elementy budowy bardziej wyrafinowanych typów danych i algorytmów języka C. Chociaż niejednokrotnie już się z nimi spotkaliśmy, ich użycie jest wystarczająco specyficzne, wyspecjalizowane i idiomatyczne, aby poświęcić im więcej uwagi. W niniejszym rozdziale zostaną omówione najczęściej stosowane sposoby użycia wskaźników, struktur i unii w programach oraz przedstawione przykłady dla każdego wzorca użycia. Rozpoznając funkcje pełnione przez określone konstrukcje językowe można lepiej zrozumieć kod, który z nich korzysta.

Umiejętność tworzenia nowych obszarów przechowywania danych w trakcie działania programu poprzez dynamiczne przydzielanie i zwalnianie pamięci stanowi jedną z istotnych koncepcji wykorzystywanych obok wskaźników i struktur w celu tworzenia i manipulowania nowymi typami danych. Wystarczy poznać kilka najważniejszych idiomów dotyczących zarządzania pamięcią przydzielaną dynamicznie, by znikła większość niejasności.

3.1. Wskaźniki

Większość osób uczących się języka C początkowo traktuje wskaźniki z pewną dozą szacunku i jednocześnie obawy. Jest prawdą, że kod wykorzystujący wskaźniki jest niekiedy bardzo tajemniczy i często stanowi źródło subtelnych błędów oraz nieprzewidzialnych awarii. Wynika to głównie z faktu, że wskaźnik jest niskopoziomowym, oferującym ogromne możliwości, ale prymitywnym narzędziem. Należy nauczyć się rozpoznawać sposoby jego użycia, co pozwoli poznać często stosowane wzorce zapisu kodu. Nieliczne pozostałe fragmenty kodu, których nie będzie można odpowiednio zaklasyfikować, należy traktować podejrzliwie.

i Wskaźniki są używane w języku C z następujących powodów:

- w celu tworzenia powiązanych struktur danych;
- w celu odwoływania się do dynamicznie przydzielanych struktur danych;
- w celu implementacji wywołań przez referencje;
- w celu uzyskiwania dostępu do elementów danych i iterowania po nich;
- w momencie przekazywania tablic jako argumentów;
- w celu odwoływania się do funkcji;
- jako aliasy innych wartości;
- w celu reprezentowania ciągów znaków;
- w celu zapewnienia bezpośredniego dostępu do pamięci systemowej.

Poniżej rozpatrzymy reprezentatywne przykłady użycia wskaźników w każdym z tych przypadków. Tam, gdzie to wskazane, zostaną zamieszczone odsyłacze do podrozdziałów, w których zawarto szczegółowe omówienie określonego przypadku użycia. Celem poniższych paragrafów jest przedstawienie podsumowania, taksonomii oraz ogólnych wytycznych dotyczących czytania kodu wykorzystującego wskaźniki.

3.1.1. Powiązane struktury danych

Na najniższym poziomie abstrakcji wskaźnik stanowi po prostu adres pamięci. Nadaje się on zatem idealnie do reprezentowania łączy między elementami struktur danych. Na poziomie koncepcyjnym struktury danych, takie jak listy, drzewa i graf, składają się z elementów (*wierzchołków*) połączonych *łukami* lub *krawędziami*. Wskaźnik może reprezentować krawędź skierowaną między dwoma wierzchołkami, przechowując w jednym z nich adres drugiego. Ponadto wierzchołki są używane podczas przetwarzania takich struktur danych w celu uzyskiwania dostępu do ich elementów oraz wykonywania operacji porządkowych. Powiązane struktury danych są omawiane w podrozdziale 4.7 i kolejnych.

3.1.2. Dynamiczne przydzielanie struktur danych

Struktury danych typu wektorowego są często przydzielane dynamicznie, tak aby ich rozmiar odpowiadał informacjom dotyczącym liczby wymaganych elementów w czasie wykonywania. Kwestię tę omówiono szczegółowo w podrozdziale 3.4. Wskaźniki są używane do przechowywania adresów początkowych takich struktur danych. Ponadto programy często dynamicznie konstruują struktury języka C w celu przekazywania ich między funkcjami lub używania jako elementów budulcowych dla powiązanych struktur danych. W takim przypadku wskaźniki są używane do przechowywania informacji o lokalizacji w pamięci przydzielonej konstrukcji `struct`, jak w poniższym przykładzie[1]:

[1] *netbsdsrc/sbin/fsck/preen.c*: 250 – 280.

```
static struct diskentry *
finddisk(const char *name)
{
   struct diskentry *d;
   [...]
   d = emalloc(sizeof(*d));
   d->d_name = estrdup(name);
   d->d_name[len] = '\0';
   [...]
   return d;
}
```

i Kod służący do przydzielenia pamięci równej co do pojemności rozmiarowi konstrukcji `struct` oraz rzutowanie wyniku na odpowiedni wskaźnik bywają często przenoszone do makra o nazwie takiej jak `new` (jak adekwatny operator języka C++)[2, 3].

```
#define new(type)    (type *) calloc(sizeof(type), 1)
    [...]
    node = new(struct codeword_entry);
```

3.1.3. Wywołania przez referencję

Wskaźniki są również używane w funkcjach, które pobierają argumenty przekazywane *przez referencję* (ang. *by reference*). Argumenty funkcji przekazywane przez referencję są wykorzystywane w celu zwracania wyników funkcji lub w celu uniknięcia narzutu czasowego związanego z kopiowaniem argumentu funkcji. Kiedy argumenty typu wskaźnikowego są używane w celu zwrócenia wyników funkcji, w jej treści znajduje się instrukcja przypisania wartości do takich argumentów, jak w przypadku poniższej funkcji, która ustawia wartość `gid` na identyfikator grupy o podanej nazwie[4].

```
int
gid_name(char *name, gid_t *gid)
{
    [...]
    *gid = ptr->gid = gr->gr_gid;
    return(0);
}
```

Wartość zwracana przez funkcję jest wykorzystywana jedynie w celu określenia błędu. Wiele funkcji interfejsu API systemu Windows wykorzystuje taką konwencję. Po stronie obiektu wywołującego zwykle widzi się odpowiedni argument przekazywany do funkcji poprzez zastosowanie operatora adresu (&) względem zmiennej.

i Argumenty wskaźnikowe są również używane w celu uniknięcia narzutu związanego z kopiowaniem dużych elementów przy każdym wywołaniu funkcji. Takie argumenty to zwykle struktury, rzadziej liczby zmiennoprzecinkowe o podwójnej precyzji. Tablice są zawsze przekazywane przez referencję i w przypadku większości architektur inne proste typy danych wydajniej jest kopiować w momencie wywołania funkcji niż przekazywać je przez referencję. Poniższa funkcja wykorzystuje dwa argumenty typu strukturalnego

[2] *XFree86-3.3/xc/doc/specs/PEX5/PEX5.1/SI/xref.h*: 113.

[3] *XFree86-3.3/xc/doc/specs/PEX5/PEX5.1/SI/xref.c*: 268.

[4] *netbsdsrc/bin/pax/cache.c*: 430 – 490.

(now oraz then) przekazywane przez referencję tylko w celu obliczenia wyniku bez modyfikowania treści struktur. Zapewne zapisano ją w ten sposób w celu uniknięcia narzutu związanego z kopiowaniem struktury timeval przy każdym wywołaniu[5].

```
static double
diffsec(struct timeval *now),
    struct timeval *then)
{
    return ((now->tv_sec - then->tv_sec)*1.0
        + (now->tv_usec - then->tv_usec)/1000000.0);
}
```

i We współczesnych programach pisanych w C i C++ często można zidentyfikować argumenty przekazywane przez referencję ze względów wydajnościowych, gdyż są opatrzone deklaratorem const[6].

```
static char *ccval (const struct cchar *, int);
```

Rola każdego z argumentów jest czasem również identyfikowana poprzez użycie komentarza IN lub OUT, jak w poniższej (zapisanej w stylu sprzed ANSI C) definicji funkcji[7]:

```
int
atmresolve(rt, m, dst, desten)
    register struct rtentry *rt;
    struct muf *m;
    register struct sockaddr *dst;
    register struct atm_pseudohdr *desten;   /* OUT */
{
```

3.1.4. Uzyskiwanie dostępu do elementów danych

W podrozdziale 4.2 zostanie przedstawiony sposób użycia wskaźnika jako kursora bazy danych służącego do uzyskiwania dostępu do elementów tabeli. Można wymienić pewne kluczowe pojęcia pomagające w czytaniu kodu, zawierające wskaźniki służące do uzyskiwania dostępu do tablic. Po pierwsze, wskaźnik na adres elementu tablicy może być wykorzystany do uzyskania dostępu do elementu znajdującego się pod określonym indeksem. Po drugie, arytmetykę wskaźników na elementy tablicy charakteryzują te same cechy semantyczne, co arytmetykę odpowiednich indeksów tablicy. W tabeli 3.1 przedstawiono przykłady wykonywania najczęściej występujących operacji w czasie uzyskiwania dostępu do tablicy a o elementach typu *T*. Na listingu 3.1[8] przedstawiono przykład kodu ze wskaźnikami służącego do uzyskiwania dostępu do stosu.

Listing 3.1. *Dostęp poprzez wskaźnik do stosu opartego na tablicy*

```
stackp = de_stack;  ●——— Inicjalizacja stosu
[...]
*stackp++ = finchar;  ●——— Odłożenie finchar na stos
[...]
do {
```

[5] *netbsdsrc/sbin/ping/ping.c*: 1028 – 1034.

[6] *netbsdsrc/bin/stty/print.c*: 56.

[7] *netbsdsrc/sys/netinet/if_atm.c*: 223 – 231.

[8] *netbsdsrc/usr.bin/compress/zopen.c*: 523 – 555.

Tabela 3.1. *Kod wykorzystujący indeksy oraz wskaźniki, służący do uzyskiwania dostępu do tablicy a o elementach typu T*

Kod z użyciem indeksów	Kod z użyciem wskaźników
`int i;`	*T* `*p`
`i = 0`	`p = a` lub `p = &a[0]`
`a[i]`	`*p`
`a[i].f`	`p->f`
`i++`	`p++`
`i += K`	`p += K`
`i == N`	`p == &a[N]` lub `p == a + N`

```
        if (count-- == 0)
            return (num);
        *bp++ = *--stackp;          <- Zdjęcie elementu ze stosu do *bp
    } while (stackp > de_stack);    <- Sprawdzenie czy stos jest pusty
```

3.1.5. Tablice jako argumenty i wyniki

W programach pisanych w językach C i C++ wskaźniki pojawiają się w przypadku przekazywania tablic do funkcji oraz ich zwracania jako wyników. W kodzie C, kiedy nazwa tablicy zostanie użyta jako argument funkcji, przekazywany jest do niej tylko adres pierwszego elementu tablicy. W rezultacie wszelkie modyfikacje dokonane na danych tablicy w czasie wykonywania funkcji mają wpływ na elementy tablicy przekazanej przez obiekt wywołujący funkcję. Takie niejawne wywołanie przez referencję w przypadku tablic odróżnia je od sposobu przekazywania do funkcji wszystkich innych typów języka C, a przez to może być źródłem niejasności.

Podobnie funkcje języka C mogą zwracać jedynie wskaźnik na element tablicy, a nie całą tablicę. A zatem, kiedy funkcja tworzy w tablicy wynik, a następnie zwraca odpowiedni wskaźnik, istotną rzeczą jest zapewnienie, aby tablica nie była zmienną lokalną, której pamięć jest przydzielana na stosie funkcji. W takim przypadku przestrzeń tablicy może zostać nadpisana po zakończeniu działania funkcji i w konsekwencji wynik jej działania będzie niepoprawny. Jednym ze sposobów na uniknięcie tego problemu jest zadeklarowanie takiej tablicy jako statycznej (`static`), jak w przypadku poniższej funkcji, która zamienia adres internetowy na jego reprezentację z wartościami dziesiętnymi rozdzielonymi kropkami[9].

```
char *inet_ntoa(struct in_addr ad)
{
  unsigned long int s_ad;
  int a, b, c, d;
  static char addr[20];

  s_ad = ad.s_addr;
  d = s_ad % 256;
  s_ad /= 256;
  c = s_ad % 256;
```

[9] *netbsdsrc/libexec/identd/identd.c*: 120 – 137.

```
    s_ad /= 256;
    b = s_ad % 256;
    a = s_ad / 256;
    sprintf(addr, "%d.%d.%d.%d", a, b, c, d);
    return addr;
}
```

Funkcja tworzy wynik w buforze `addr`. Gdyby nie został on zadeklarowany jako `static`, jego zawartość (czytelna reprezentacja adresu internetowego) stałaby się nieprawidłowa po zakończeniu działania funkcji. Nawet powyższa konstrukcja nie jest w pełni bezpieczna. Funkcje wykorzystujące globalne lub statyczne zmienne lokalne nie są w większości przypadków wielobieżne. Oznacza to, że funkcja nie może zostać wywołana z poziomu wątku innego programu, kiedy jedna instancja tej funkcji już działa. Co gorsza, w naszym przypadku wynik funkcji musi zostać zapisany w innym miejscu (na przykład przy użyciu wywołania funkcji `strdup`) zanim funkcja zostanie wywołana ponownie. W przeciwnym razie zostałby on nadpisany nowym rezultatem. Przykładowo, przedstawiona implementacja funkcji `inet_ntoa` nie mogłaby zostać użyta zamiast funkcji `naddr_ntoa` w następującym kontekście[10]:

```
(void)fprintf(ftrace, "%s Router Ad from %s to %s via %s life=%d\n",
         act, naddr_ntoa(from), naddr_ntoa(to),
         ifp ? ifp->int_name : "?",
         ntohs(p->ad.icmp_ad_life));
```

W tym przypadku w celu obejścia opisanego problemu funkcja `naddr_ntoa` jest używana jako kod opakowujący dla funkcji `inet_ntoa`, zapewniając przechowanie jej wyniku w liście cyklicznej o czterech różnych buforach tymczasowych[11].

```
char *
naddr_ntoa(naddr a)
{
#define NUM_BUFS 4
    static int bufno;
    static struct {
        char    str[16];      /* xxx.xxx.xxx.xxx\0 */
    } bufs[NUM_BUFS];
    char *s;
    struct in_addr addr;

    addr.s_addr = a;
    s = strcpy(bufs[bufno].str, inet_ntoa(addr));
    bufno = (bufno+1) % NUM_BUFS;
    return s;
}
```

3.1.6. Wskaźniki na funkcje

Często przydatnym rozwiązaniem jest sparametryzowanie funkcji poprzez przekazywanie jej do innej funkcji jako argumentu. Jednak język C nie dopuszcza przekazywania funkcji jako argumentów. Możliwe jest jednak przekazanie jako argumentu *wskaźnika* na funkcję. Na listingu 3.2[12] funkcja `getfile`, używana do przetwarzania plików w czasie procesu odtwarzania po awarii, pobiera parametry `fill` oraz `skip`. Służą one do

[10] *netbsdsrc/sbin/routed/rdisc.c*: 121 – 125.

[11] *netbsdsrc/sbin/routed/trace.c*: 123 – 139.

[12] *netbsdsrc/sbin/restore/tape.c*: 177 – 837.

określenia, w jaki sposób należy odczytywać lub pomijać dane. Funkcja jest wywoływana (listing 3.2:1) z różnymi argumentami (listing 3.2:3) w celu wykonania początkowego przeglądu danych lub odtworzenia albo pominięcia plików danych.

Listing 3.2. *Parametryzacja przy użyciu argumentów funkcji*

```
/*
 * Verify that the tape drive can be accessed and
 * that it actually is a dump tape.
 */
void
setup()
{
    [...]
    getfile(xtrmap, xtrmapskip);
    [...]
}

/* Prompt user to load a new dump volume. */
void
getvol(int nextvol)
{
    [...]
    getfile(xtrlnkfile, xtrlnkskip);
    [...]
    getfile(xtrfile, xtrskip);
    [...]
}

/*
 * skip over a file on the tape
 */
void
skipfile()
{
    curfile.action = SKIP;
    getfile(xtrnull, xtrnull);
}

/* Extract a file from the tape. */
void
getfile(void (*fill)(char *, long), (*skip)(char *, long))
{
    [...]
    (*fill)((char *)buf, (long)(size > TP_BSIZE ?
        fssize : (curblk - 1) * TP_BSIZE + size));
    [...]
    (*skip)(clearedbuf, (long)(size > TP BSIZE ?
        TP_BSIZE : size));
    [...]
    (*fill)((char *)buf, (long)((curblk * TP_BSIZE) + size));
    [...]
}

/* Write out the next block of a file. */
static void
xtrfile(char *buf, long size)
{ [...] }
/* Skip over a hole in a file. */
static void
xtrskip(char *buf, long size)
{ [...] }
/* Collect the next block of a symbolic link. */
static void
```

[1] Wywołanie z argumentami będącymi wskaźnikami na funkcje

[1]

[1]

[1]

[2] Wywołanie funkcji przekazanej w argumencie

[2]

[2]

[3] Funkcje przekazywane jako parametry argumentów

```
xtrlnkfile(char *buf, long size)
{ [...] }
/* Skip over a hole in a symbolic link (should never happen). */
static void
xtrlnkskip(char *buf, long size)
{ [...] }
/* Collect the next block of a bit map. */
static void
xtrmap(char *buf, long size)
{ [...] }
/* Skip over a hole in a bit map (should never happen). */
static void
xtrmapskip(char *buf, long size)
{ [...] }
/* Noop, when an extraction function is not needed. */
void
xtrnull(char *buf, long size)
{ return; }
```

Wiele plików z bibliotek języka C, takich jak `qsort` i `bsearch`, pobiera argumenty będące wskaźnikami na funkcje, które określają sposób ich działania[13].

```
getnfile()
{
    [...]
    qsort(nl, nname, sizeof(nltype), valcmp);
    [...]
}

valcmp(nltype *p1, nltype *p2)
{
    if ( p1 -> value < p2 -> value ) {
        return LESSTHAN;
    }
    if ( p1 -> value > p2 -> value ) {
        return GREATERTHAN;
    }
    return EQUALTO;
}
```

W powyższym przykładzie sposób, w jaki funkcja `qsort` porównuje elementy sortowanej tablicy, określa funkcja `valcmp`.

Wreszcie, wskaźniki na funkcje mogą być używane do parametryzowania sterowania w treści kodu. W poniższym przykładzie wskaźnik `closefunc` jest używany do przechowywania funkcji, która zostanie wywołana w celu zamknięcia strumienia `fin` w zależności od sposobu jego otwarcia[14].

```
void
retrieve(char *cmd, char *name)
{
    int (*closefunc)(FILE *) = NULL;
    [...]
    if (cmd == 0) {
        fin = fopen(name, "r"), closefunc = fclose;
    [...]
    if (cmd) {
```

[13] *netbsdsrc/usr.bin/gprof/gprof.c*: 216 – 536.

[14] *netbsdsrc/libexec/ftpd/ftpd.c*: 792 – 860.

```
        [...]
        fin = ftpd_popen(line, "r", 1), closefunc = ftpd_pclose;
    [...]
   (*closefunc)(fin);
}
```

3.1.7. Wskaźniki jako aliasy

Wskaźniki są często używane do tworzenia *aliasów* pewnych wartości[15].

```
struct output output = {NULL, 0, NULL, OUTBUFSIZ, 1, 0};
[...]
struct output *out1 = &output;
```

W powyższym przykładzie kodu poddany dereferencji wskaźnik `out1` może zostać użyty zamiast oryginalnej wartości zmiennej `output`. Aliasów używa się z wielu różnych powodów.

Kwestie wydajnościowe

Przypisanie wskaźnika jest bardziej wydajne od przypisania większego obiektu. W poniższym przykładzie wskaźnik `curt` mógłby być zmienną strukturalną, a nie wskaźnikiem na taką strukturę. Jednak odpowiednia instrukcja przypisania byłaby mniej wydajna, gdyż wiązałaby się ze skopiowaniem zawartości całej struktury[16].

```
static struct termios cbreakt, rawt, *curt;
    [...]
    curt = useraw ? &rawt : &cbreakt;
```

Odwołania do statycznie inicjalizowanych danych

Zmienna jest używana do wskazywania na różne wartości danych statycznych. Najczęściej spotykany przykład w tym przypadku to zbiór wskaźników znakowych wskazujących na różne ciągi znaków[17].

```
char *s;
[...]
s = *(opt->bval) ? "True" : "False";
```

Implementacja semantyki odwołań do zmiennych w kontekście globalnym

Mało przyjazny tytuł niniejszego punktu odnosi się do odpowiednika użycia wskaźników wywołań przez referencję, tyle że z wykorzystaniem zmiennej globalnej zamiast argumentu funkcji. Zatem globalna zmienna wskaźnikowa jest używana do odwoływania się do danych, do których należy uzyskać dostęp i zmodyfikować je w innym miejscu. W przedstawionym poniżej generatorze liczb pseudolosowych wskaźnik `fptr` jest

[15] *netbsdsrc/bin/sh/output.c*: 81 – 84.

[16] *netbsdsrc/lib/libcurses/tty.c*: 66 – 171.

[17] *netbsdsrc/games/rogue/room.c*: 629 – 635.

inicjalizowany tak, aby wskazywał na wpis w tabeli ziaren generatora liczb pseudolosowych. W podobny sposób jest on ustawiany w funkcji `srrandom()`. W końcu, w funkcji `rrandom()` zmienna `fptr` jest używana do zmodyfikowania wartości, na którą wskazuje[18].

```
static long rntb[32] = {
    3, 0x9a319039, 0x32d9c024, 0x9b663182, 0x5da1f342,
    [...]
};

static long *fptr = &rntb[4];
[...]

void
srrandom(int x)
{
    [...]

    fptr = &state[rand_sep];
    [...]
}

long
rrandom()
{
    [...]
    *fptr += *rptr;
    i = (*fptr >> 1) & 0x7fffffff;
```

Równoważny rezultat otrzymalibyśmy, przekazując `fptr` jako argument do funkcji `srrandom()` oraz `rrandom()`.

Podobne podejście jest często wykorzystywane w celu uzyskiwania dostępu do modyfikowalnej kopii danych globalnych[19].

```
WINDOW scr_buf;
WINDOW *curscr = &scr_buf;
[...]
move(short row, short col);
{
    curscr->_cury = row;
    curscr->_curx = col;
    screen_dirty = 1;
}
```

3.1.8. Wskaźniki a ciągi znaków

W języku C literały znakowe są reprezentowane przez tablicę znaków zakończoną znakiem zerowym `'\0'`. W rezultacie ciągi znaków są reprezentowane przez wskaźnik na pierwszy znak sekwencji zakończonej znakiem zerowym. W poniższym fragmencie kodu można zobaczyć, w jaki sposób kod funkcji bibliotecznej języka C `strlen` przesuwa wskaźnik wzdłuż ciągu znaków przekazanego jako parametr funkcji do momentu osiągnięcia jego końca. Następnie odejmuje wskaźnik na początek ciągu od wskaźnika na końcowy znak zerowy w celu obliczenia i zwrócenia długości ciągu[20].

[18] *netbsdsrc/games/rogue/random.c*: 62 – 109.

[19] *netbsdsrc/games/rogue/curses.c*: 121 – 157.

[20] *netbsdsrc/lib/libc/string/strlen.c*: 51 – 59.

```
size_t
strlen(const char *str)
{
    register const char *s;

    for (s = str; *s; ++s)
        ;
    return(s - str);
}
```

⚠ Czytając kod operujący na ciągach znaków, należy pamiętać o istniejącym rozróżnieniu na wskaźniki znakowe i tablice znaków. Choć obie konstrukcje są często używane do reprezentowania ciągów znaków (ze względu na fakt, że tablica znaków jest automatycznie konwertowana na wskaźnik na pierwszy znak w tablicy w momencie przekazania do funkcji), odpowiednie typy i operacje, które można na nich wykonywać, są różne. Przykładowo, poniższy kod definiuje zmienną `pw_file` jako wskaźnik na znak wskazujący na stałą znakową zawierającą sekwencję `"/etc/passwd"`[21].

```
static char *pw_file = "/etc/passwd";
```

Rozmiar zmiennej `pw_file` na maszynie Autora wynosi 4 i może ona zostać zmodyfikowana tak, aby wskazywała inne miejsce, jednak próba zmiany zawartości pamięci, na którą wskazuje, spowoduje niezdefiniowane zachowanie.

Z drugiej strony, poniższy wiersz definiuje zmienną `line` jako tablicę znaków zainicjalizowaną treścią `"/dev/XtyXX"`, po której występuje wartość zerowa[22].

```
static char line[] = "/dev/XtyXX";
```

Zastosowanie operatora `sizeof` względem `line` daje w wyniku 11. Zmienna `line` zawsze będzie odwoływać się do tego samego obszaru pamięci, a elementy które zawiera mogą być dowolnie modyfikowane[23].

```
line[5] = 'p';
```

Pamiętanie o różnicach istniejących między tablicami znaków a wskaźnikami podczas czytania kodu jest istotne, gdyż obie konstrukcje są używane, często zamiennie, w podobnych celach. W szczególności żaden kompilator języka C nie ostrzega użytkownika o sytuacji, w której obie zostaną przypadkowo wymieszane w ramach różnych plików. Weźmy pod uwagę poniższe dwie (niezwiązane ze sobą) definicje oraz odpowiednią deklarację[24, 25, 26].

```
char version[] = "332";
char *version = "2.1.2";
extern char *version;
```

[21] *netbsdsrc/distrib/utils/libhack/getpwent.c*: 45.

[22] *netbsdsrc/lib/libutil/pty.c*: 71.

[23] *netbsdsrc/lib/libutil/pty.c*: 81.

[24] *netbsdsrc/usr.bin/less/less/version.c*: 575.

[25] *netbsdsrc/libexec/identd/version.c*: 3.

[26] *netbsdsrc/libexec/identd/identd.c*: 77.

Obie definicje są używane do zdefiniowania ciągu informującego o wersji, który jest zapewne przekazywany do funkcji `printf` w celu wydrukowania na ekranie. Jednakże deklaracja `extern char *version` może być użyta jedynie w celu uzyskania dostępu do zmiennej zdefiniowanej jako `char *version`. Choć konsolidacja pliku źródłowego z plikiem zawierającym definicję `char version[]` zwykle nie spowoduje wygenerowania błędu, wynikowy program zakończy awaryjnie działanie po uruchomieniu. Zmienna `version` zamiast wskazywać na obszar pamięci, zawierający ciąg z informacją o wersji, będzie prawdopodobnie wskazywała na obszar pamięci, której adres jest reprezentowany przez wzorzec bitowy ciągu znaków `"332"` (`0x33333200` w przypadku architektur niektórych procesorów).

3.1.9. Bezpośredni dostęp do pamięci

Ze względu na fakt, że wskaźniki są wewnętrznie reprezentowane jako adresy pamięci, można spotkać kod niskopoziomowy wykorzystujący wskaźniki do uzyskiwania dostępu do obszarów pamięci związanych z zasobami sprzętowymi. Wiele urządzeń peryferyjnych, takich jak karty graficzne i sieciowe, wykorzystuje ogólnodostępną pamięć w celu wymiany danych z procesorem systemowym. Zmienna zainicjalizowana tak, aby wskazywała na ten obszar, może być z łatwością wykorzystywana do komunikowania się z danym urządzeniem. W poniższym przykładzie zmienna `video` jest ustawiana tak, aby wskazywała na obszar pamięci zajmowany przez sterownik ekranu (`0xe08b8000`). Zapis do tego obszaru przy przesunięciu określonym przez dany wiersz i kolumnę (`vid_ypos` oraz `vid_xpos`) powoduje pojawienie się na ekranie odpowiedniego znaku (`c`)[27].

```
static void
vid_wrchar(char c)
{
    volatile unsigned short *video;

    video = (unsigned short *)(0xe08b8000) + vid_ypos * 80 + vid_xpos;
    *video = (*video & 0xff00) | 0x0f00 | (unsigned short)c;
}
```

Należy pamiętać, że współczesne systemy operacyjne zapobiegają uzyskiwaniu dostępu do zasobów sprzętowych przez programy użytkownika bez podjęcia wcześniej odpowiednich kroków. Tego rodzaju kod spotyka się w zasadzie tylko w przypadku badania systemów osadzonych lub kodu jądra systemu i sterowników urządzeń.

Ćwiczenie 3.1. Zaproponuj własną implementację stosu opartego na wskaźnikach z biblioteki *compress*[28] używając indeksu tablicy. Zmierz różnicę w szybkości i skomentuj czytelność obu implementacji.

Ćwiczenie 3.2. Zlokalizuj na płycie CD-ROM dołączonej do książki trzy wystąpienia kodu używającego wskaźników z każdego z wymienionych powodów.

[27] *netbsdsrc/sys/arch/pica/pica/machdep.c*: 951 – 958.

[28] *netbsdsrc/usr.bin/compress/zopen.c*

Ćwiczenie 3.3. Jeżeli Czytelnik zna język C++ lub Java, może spróbować wyjaśnić, w jaki sposób można zminimalizować (w języku C++) lub uniknąć (w języku Java) używania wskaźników.

Ćwiczenie 3.4. Czym różni się wskaźnik języka C od adresu pamięci? Jaki ma to wpływ na zrozumienie kodu? Jakie narzędzia wykorzystują istniejącą różnicę?

Ćwiczenie 3.5. Czy ciągi znaków zawierające informacje o wersji programu powinny być reprezentowane jako tablice znaków czy jako wskaźniki na ciągi znaków? Uzasadnij odpowiedź.

3.2. Struktury

i Konstrukcja języka C `struct` to zgrupowanie elementów danych, które umożliwia używanie ich wspólnie. Struktury są używane w programach pisanych w C w celu:

- zgrupowania elementów danych używanych zwykle wspólnie;
- zwrócenia wielu elementów danych z funkcji;
- konstruowania powiązanych struktur danych (podrozdział 4.7);
- odwzorowania organizacji danych w urządzeniach sprzętowych, łączach sieciowych oraz nośnikach danych;
- implementacji abstrakcyjnych typów danych (podrozdział 9.3.5);
- tworzenia programów zgodnie z paradygmatem obiektowym.

Poniższe punkty stanowią rozwinięcie dyskusji na temat użycia struktur i poruszają zagadnienia nie omawiane w innych podrozdziałach.

3.2.1. Grupowanie elementów danych

Struktury są często używane w celu tworzenia grup powiązanych elementów, które zazwyczaj są używane wspólnie jako całość. Sztandarowym przykładem jest tu reprezentacja współrzędnych pozycji na ekranie[29].

```
struct point {
    int col, line;
};
```

W innych przypadkach może chodzić o liczby zespolone lub pola tworzące wiersze tabeli.

3.2.2. Zwracanie wielu elementów danych z funkcji

i Kiedy wynik działania funkcji należy wyrazić używając więcej niż jednego podstawowego typu danych, wiele elementów wyniku można zwrócić albo poprzez argumenty wywołania funkcji przekazane przez referencję (patrz podrozdział 3.1.3), albo grupując

[29] *netbsdsrc/games/snake/snake/snake.c*: 75 – 77.

je w ramach zwracanej struktury. W poniższym przykładzie funkcja `difftv` zwraca różnicę między dwiema wartościami czasowymi wyrażonymi w sekundach (`tv_sec`) oraz w mikrosekundach (`tv_usec`) w postaci struktury `timeval`[30].

```
static struct timeval
difftv(struct timeval a, struct timeval b)
{
    static struct timeval diff;

    diff.tv_sec = b.tv_sec - a.tv_sec;
    if ((diff.tv_usec = b.tv_usec - a.tv_usec) < 0) {
        diff.tv_sec--;
        diff.tv_usec += 1000000;
    }
    return(diff);
}
```

3.2.3. Odwzorowanie organizacji danych

i Kiedy dane są przesyłane przez sieć, przenoszone z i na urządzenie pamięci masowej lub kiedy programy bezpośrednio komunikują się ze sprzętem, struktury są często używane w celu reprezentowania sposobu organizacji danych w przypadku takiego medium. Poniższa struktura reprezentuje blok poleceń karty sieciowej Intel EtherExpress[31].

```
struct fxp_cb_nop {
    void *fill[2];
    volatile u_int16_t cb_status;
    volatile u_int16_t cb_command;
    volatile u_int16_t link_addr;
};
```

i Kwalifikator `volatile` jest używany w celu oznaczenia, że odpowiednie pola pamięci są używane przez obiekty pozostające poza kontrolą programu (w tym przypadku chodzi o kartę sieciową). Kompilator nie może więc przeprowadzać optymalizacji na takich polach, na przykład usuwać nadmiarowych referencji.

i W określonych przypadkach można zadeklarować *pole bitowe* (ang. *bit field*), określające ścisły zakres bitów używanych do przechowywania określonej wartości w danym urządzeniu[32].

```
struct fxp_cb_config {
    [...]
    volatile u_int8_t   byte_count:6,
                :2;
    volatile u_int8_t   rx_fifo_limit:4,
                tx_fifo_limit:3,
                :1;
```

W powyższym przykładzie liczba przesyłanych bajtów określona jako `byte_count` ma zajmować 6 bitów, natomiast ograniczenia kolejek FIFO odbiornika i nadajnika mają zajmować, odpowiednio, 4 i 3 bity w urządzeniu sprzętowym.

[30] *netbsdsrc/usr.bin/named/dig/dig.c*: 1221 – 1233.

[31] *netbsdsrc/sys/dev/pci/if_fxpreg.h*: 102 – 107.

[32] *netbsdsrc/sys/dev/pci/if_fxpreg.h*: 116 – 125.

Pakiety danych sieciowych również są często kodowane przy użyciu struktur języka C w celu określenia struktury ich elementów, co ukazuje poniższa klasyczna definicja nagłówka pakietów TCP[33].

```
struct tcphdr {
    u_int16_t th_sport;      /* source port */
    u_int16_t th_dport;      /* destination port */
    tcp_seq     th_seq;      /* sequence number */
    tcp_seq     th_ack;      /* acknowledgement number */
```

Wreszcie, struktury są również używane w celu odwzorowania sposobu przechowywania danych na nośnikach danych, na przykład dyskach lub taśmach. Przykładowo, właściwości dysku partycji systemu MS-DOS określa się za pomocą tak zwanego bloku parametrów BIOS. Jego pola odwzorowuje poniższa struktura[34].

```
struct bpb33 {
    u_int16_t   bpbBytesPerSec;  /* bytes per sector */
    u_int8_t    bpbSecPerClust;  /* sectors per cluster */
    u_int16_t   bpbResSectors;   /* number of reserved sectors */
    u_int8_t    bpbFATs;         /* number of FATs */
    u_int16_t   bpbRootDirEnts;  /* number of root directory entries */
    u_int16_t   bpbSectors;      /* total number of sectors */
    u_int8_t    bpbMedia;        /* media descriptor */
    u_int16_t   bpbFATsecs;      /* number of sectors per FAT */
    u_int16_t   bpbSecPerTrack;  /* sectors per track */
    u_int16_t   bpbHeads;        /* number of heads */
    u_int16_t   bpbHiddenSecs;   /* number of hidden sectors */
};
```

⚠ Sposób uporządkowania pól w ramach struktury jest zależny od architektury oraz kompilatora. Ponadto reprezentacja różnych elementów w strukturze jest zależna od architektury oraz systemu operacyjnego (system operacyjny może wymuszać na procesorze dostosowanie się do określonej kolejności przechowywania bajtów). Nawet proste typy danych, takie jak liczby całkowite, mogą posiadać swoją wartość przechowywaną na różne sposoby. Stąd użycie struktur w celu odwzorowania zewnętrznych danych stanowi z gruntu nieprzenośne rozwiązanie.

3.2.4. Programowanie obiektowe

i W języku C struktury są czasem używane do tworzenia konstrukcji przypominających obiekty poprzez zgrupowanie razem elementów danych i wskaźników na funkcje w celu zasymulowania pól i metod klasy. W poniższym przykładzie struktura `domain`, reprezentująca różne domeny protokołu sieciowego (na przykład internet, SNA, IPX), grupuje dane dotyczące określonej domeny, takie jak jej rodzina `dom_family` oraz metody służące do operowania na nich, takie jak metoda inicjalizacji tabeli routingu `dom_rtattach`[35].

```
struct   domain {
    int    dom_family;         /* AF_xxx */
    char   *dom_name;
```

[33] *netbsdsrc/sys/netinet/tcp.h*: 43 – 47.

[34] *netbsdsrc/sys/msdosfs/bpb.h*: 22 – 34.

[35] *netbsdsrc/sys/sys/domain.h*: 50 – 65.

```
    void   (*dom_init)(void);  /* initialize domain data structures */
    [...]
    int   (*dom_rtattach)(void **, int);
                            /* initialize routing table */
    int   dom_rtoffset;     /* an arg to rtattach, in bits */
    int   dom_maxrtkey;     /* for routing layer */
};
```

Po wprowadzeniu takiej deklaracji, zmienne określonego typu strukturalnego, czy też ściślej rzecz ujmując — wskaźniki na typ strukturalny — po odpowiednim zainicjalizowaniu mogą być traktowane w sposób przypominający użycie obiektów w językach C++ i Java[36].

```
for (dom = domains; dom; dom = dom->dom_next)
    if (dom->dom_family == i && dom->dom_rtattach) {
        dom->dom_rtattach((void **)&nep->ne_rtable[i],
            dom->dom_rtoffset);
        break;
```

Ze względu na fakt, że „obiekty" mogą być inicjalizowane za pomocą różnych „metod" (wskaźników na funkcje), a jednak są używane poprzez ten sam interfejs (wywołania następują poprzez te same nazwy składowych struktur), opisana technika stanowi implementację *metod wirtualnych* (ang. *virtual methods*), dostępnych w językach obiektowych, oraz *programowania polimorficznego* (ang. *polymorphic programming*). W rzeczywistości, ze względu na fakt, że obiekty należące do tej samej klasy współużytkują swoje metody (ale nie pola), wskaźniki na metody są często dzielone przez różne obiekty dzięki przechowywaniu w strukturze „obiektów" jedynie wskaźnika na inną strukturę zawierającą wskaźniki na faktyczne metody[37].

```
struct file {
    [...]
    short   f_type;          /* descriptor type */
    short   f_count;         /* reference count */
    short   f_msgcount;      /* references from message queue */
    struct   ucred *f_cred;  /* credentials associated with descriptor */
    struct   fileops {
        int   (*fo_read)(struct file *fp, struct uio *uio,
                                struct ucred *cred);
        int   (*fo_write)(struct file *fp, struct uio *uio,
                                struct ucred *cred);
        int   (*fo_ioctl)(struct file *fp, u_long com,
                                caddr_t data, struct proc *p);
        int   (*fo_poll)(struct file *fp, int events,
                                struct proc *p);
        int   (*fo_close)(struct file *fp, struct proc *p);
    } *f_ops;
    off_t   f_offset;
    caddr_t   f_data;      /* vnode or socket */
};
```

W powyższym przykładzie każdy obiekt reprezentujący otwarty plik lub gniazdo dzieli swoje metody `read`, `write`, `ioctl`, `poll` oraz `close` poprzez składową struktury `f_ops`, wskazującą na dzieloną strukturę `fileops`.

[36] *netbsdsrc/sys/kern/vfs_subr.c*: 1436 – 1440.

[37] *netbsdsrc/sys/sys/file.h*: 51 – 73.

Ćwiczenie 3.6. Alternatywnym sposobem przekazywania do funkcji danych typu tablicowego przez wartość i zwracania ich jako rzeczywistych wartości jest zawarcie tablicy w strukturze. Zlokalizuj takie przykłady na płycie dołączonej do książki. Wyjaśnij, dlaczego takie podejście nie jest wykorzystywane zbyt często.

Ćwiczenie 3.7. Zlokalizuj 20 oddzielnych wystąpień struktur na płycie dołączonej do książki i określ powód ich wykorzystania. Zapoznaj się z biblioteką struktur danych ogólnego przeznaczenia, taką jak STL języka C++ lub `java.util`. Wskaż, które wystąpienia można by zapisać korzystając z biblioteki. Zminimalizuj czas poświęcony każdemu wystąpieniu.

Ćwiczenie 3.8. Wiele technik, bibliotek oraz narzędzi obsługuje przenośne kodowanie danych w celu ich przenoszenia między aplikacjami. Określ techniki mające zastosowanie w Twoim środowisku i porównaj je z użyciem podejścia bazującego na strukturach języka C.

3.3. Unie

Konstrukcja języka C `union` grupuje elementy, które dzielą ten sam obszar pamięci. Możliwy jest dostęp tylko do jednego elementu naraz spośród współużytkujących taki obszar. Unie są używane w języku C w celu:

- zapewnienia wydajnego wykorzystania pamięci;
- zaimplementowania polimorfizmu;
- umożliwienia dostępu do danych przy użyciu różnych reprezentacji wewnętrznych.

3.3.1. Wydajne wykorzystanie pamięci

Często spotykane uzasadnienie wykorzystania unii dotyczy współużytkowania tego samego obszaru pamięci w dwóch różnych celach. Ma to na celu zaoszczędzenie przynajmniej kilku bajtów pamięci. Istnieją przypadki, w których unie są wykorzystywane wyłącznie w tym celu. Jednak w przypadku, gdy urządzenia wbudowane obsługują pamięć o wielomegabajtowej pojemności, używanie unii staje się nieuzasadnione. Oprócz starszego kodu można również spotkać przypadki, gdy duża liczba obiektów opartych na unii uzasadnia dodatkowe wysiłki związane z napisaniem odpowiedniego kodu. Poniższy przykład pochodzący z funkcji `malloc` standardowej biblioteki języka C to typowy przypadek[38].

```
union   overhead {
    union   overhead *ov_next; /* when free */
    struct {
      u_char   ovu_magic;      /* magic number */
      u_char   ovu_index;      /* bucket # */
#ifdef RCHECK
      u_short  ovu_rmagic;     /* range magic number */
```

[38] *netbsdsrc/lib/libc/stdlib/malloc.h*: 78 – 92.

```
        u_long   ovu_size;       /* actual block size */
#endif
    } ovu;
#define   ov_magic   ovu.ovu_magic
#define   ov_index   ovu.ovu_index
#define   ov_rmagic  ovu.ovu_rmagic
#define   ov_size    ovu.ovu_size
};
```

Bloki pamięci mogą być zajęte lub wolne. W każdym przypadku muszą być przechowywane różne wartości a ze względu na fakt, że blok nie może być jednocześnie wolny i zajęty, mogą one dzielić tę samą przestrzeń pamięci. Dodatkowe koszty programistyczne związane z konserwacją takiego rozwiązania są amortyzowane tysiącami elementów, które są przydzielane w wyniku wywołań funkcji bibliotecznych.

i Warto zwrócić uwagę na definicję makro, która występuje po definicji struktury `ovu`. Takie definicje są często używane jako skrócona forma odwoływania się bezpośrednio do składowych struktury w ramach unii bez konieczności używania na początku nazwy składowej unii. Stąd przedstawiony kod ma postać[39]

```
op->ov_index = bucket;
```

zamiast bardziej rozbudowanego odwołania `op->ovu.ovu_index`.

3.3.2. Implementacja polimorfizmu

Najczęściej występującym powodem wykorzystania unii jest chęć zaimplementowania polimorfizmu. W tym przypadku ten sam obiekt (zwykle reprezentowany przez struktu-
i rę języka C) jest używany w celu reprezentowania różnych typów. Dane dla takich różnych typów są przechowywane w oddzielnych składowych unii. Dane polimorficzne przechowywane w ramach unii pozwalają również zaoszczędzić pamięć w przypadku różnych konfiguracji. Jednak w tym przypadku unia jest używana w celu wyrażenia charakterystyki obiektu, który przechowuje różne typy danych, a nie zaoszczędzenia
i zasobów. Unie używane w ten sposób są zwykle zawarte w strukturze zawierającej pole typu, które określa, jakiego rodzaju dane są przechowywane w unii. To pole jest często reprezentowane za pomocą typu wyliczeniowego, a jego nazwą jest zwykle `type`. Poniższy przykład, część biblioteki RPC (ang. *Remote Procedure Call*; zdalne wywoływanie procedur), zawiera strukturę używaną do reprezentowania komunikatów RPC. Obsługiwane są dwa różne rodzaje komunikatów: wywołanie oraz odpowiedź. Typ wyliczeniowy `msg_type` stanowi rozróżnienie między tymi dwoma typami, natomiast unia zawiera strukturę z elementami danych dla każdego typu[40].

```
enum msg_type {
    CALL=0,
    REPLY=1
};
[...]
struct rpc_msg {
    u_int32_t      rm_xid;
    enum msg_type      rm_direction;
```

[39] *netbsdsrc/lib/libc/stdlib/malloc.c*: 213.

[40] *netbsdsrc/include/rpc/rpc_msg.h*: 54 – 158.

```
    union {
        struct call_body RM_cmb;
        struct reply_body RM_rmb;
    } ru;
};
```

3.3.3. Uzyskiwanie dostępu do różnych reprezentacji wewnętrznych

Ostatnie użycie unii jest związane z przechowywaniem danych w ramach jednego pola unii oraz uzyskiwaniem dostępu do innego w celu przeniesienia danych między różnymi reprezentacjami wewnętrznymi. Choć tego rodzaju rozwiązania są z gruntu nieprzenośne, można bez przeszkód przeprowadzać pewne konwersje. Inne są przydatne w pewnych określonych przypadkach, zależnych od danej maszyny. Poniższa definicja struktury jest używana przez program archiwizujący *tar* w celu reprezentowania informacji o każdym pliku z archiwum[41].

```
union record {
    char      charptr[RECORDSIZE];
    struct header {
        char   name[NAMSIZ];
        char   mode[8];
        char   uid[8];
        char   gid[8];
        char   size[12];
        char   mtime[12];
        char   chksum[8];
        char   linkflag;
        char   linkname[NAMSIZ];
        char   magic[8];
        char   uname[TUNMLEN];
        char   gname[TGNMLEN];
        char   devmajor[8];
        char   devminor[8];
    } header;
};
```

Aby umożliwić wykrywanie uszkodzenia danych, w polu `chksum` jest umieszczana suma bajtów wszystkich rekordów składających się na plik (wliczając w to rekord nagłówkowy). Programy wykorzystują składową unii `charptr` do iteracyjnego przeglądania danych nagłówka bajt po bajcie, obliczając sumę kontrolną oraz składową `header` w celu uzyskania dostępu do określonych pól nagłówka. Ze względu na fakt, że typy całkowite w języku C (wliczając znaki) są poprawne w zakresie wszystkich możliwych wzorców bitowych, jakie mogą reprezentować, uzyskiwanie dostępu do wewnętrznej reprezentacji innych typów języka C (wskaźników, liczb zmiennoprzecinkowych i innych typów całkowitych) jako typu całkowitego jest z założenia operacją dozwoloną. Jednak operacja odwrotna — generowanie innego typu poprzez jego reprezentację całkowitą — nie zawsze daje poprawne wyniki. W przypadku większości architektur operacją bezpieczną jest generowanie typów niecałkowitych na podstawie danych, które stanowią starszą wersję kopii ich wartości.

i

[41] *netbsdsrc/usr.bin/file/tar.h*: 36 – 54.

Poza tym wykorzystanie struktur w celu uzyskania dostępu do zależnych od architektury elementów danych w pewnym innym formacie jest działaniem z gruntu nieprzenośnym. Może to być przydatne w przypadku interpretowania typu danych w oparciu o jego reprezentację lub tworzenia typu danych na podstawie jego reprezentacji. W przykładzie z listingu 3.3[42] unia `u` jest używana w celu uzyskania dostępu do wewnętrznej reprezentacji liczby zmiennoprzecinkowej `v` w formie mantysy, wykładnika oraz znaku. Taka konwersja jest wykorzystywana do rozbicia liczby zmiennoprzecinkowej na znormalizowany ułamek oraz całkowitą potęgę liczby 2.

Listing 3.3. *Uzyskiwanie dostępu do wewnętrznej reprezentacji typu przez wykorzystanie unii*

```
double
frexp(double value, int *eptr)          ●—— Zwracany wykładnik
{
    union {
        double v;                       ●—— Wartość jest przechowywana w tym polu
                                        ●—— Dostęp do reprezentacji wewnętrznej odbywa się przez to pole
        struct {
            u_int u_mant2 : 32;         ●—— Mantysa
            u_int u_mant1 : 20;
            u_int   u_exp : 11;         ●—— Wykładnik
            u_int  u_sign :  1;
        } s;

    } u;
    if (value) {
        u.v = value;                    ●—— Zachowanie wartości
        *eptr = u.s.u_exp - 1022;       ●—— Pobranie i ustawienie wykładnika ze znakiem
        u.s.u_exp = 1022;               ●—— Wykładnik zerowy
        return(u.v);                    ●—— Zwrócenie znormalizowanej mantysy
    } else {
        *eptr = 0;
        return((double)0);
    }
}
```

Ćwiczenie 3.9. Zlokalizuj 20 różnych wystąpień unii na płycie dołączonej do książki i sklasyfikuj powody ich wykorzystania. Zminimalizuj czas poświęcony każdemu wystąpieniu. Utwórz wykres ilustrujący częstotliwość używania tej konstrukcji w zależności od powodu.

Ćwiczenie 3.10. Zaproponuj przenośną alternatywę względem implementacji nieprzenośnych konstrukcji używanych w przypadku struktur i unii. Omów swoją propozycję w kontekście kosztów implementacyjnych, możliwości konserwacji oraz wydajności.

3.4. Dynamiczne przydzielanie pamięci

Struktura danych, której rozmiar nie jest znany w momencie pisania programu lub wzrasta w podczas pracy programu, jest przechowywana w pamięci przydzielanej *dynamicznie* w trakcie jego działania. Programy odwołują się do pamięci przydzielanej dynamicznie dzięki użyciu wskaźników. W niniejszym podrozdziale zostaną przedstawione spo-

[42] *netbsdsrc/lib/libc/arch/i386/gen/frexp.c*: 48 – 72.

soby dynamicznego przydziału pamięci w przypadku struktur wektorowych. Jednak prezentowane fragmenty kodu są bardzo podobne lub identyczne z tymi, które służą do przechowywania innych struktur danych.

Na listingu 3.4[43] przedstawiono typowy przykład sposobu dynamicznego przydzielania i używania przestrzeni danych. W tym przypadku pamięć zostaje przydzielona w celu przechowywania ciągu liczb całkowitych w formie tablicy, tak więc na początku zmienna `RRlen` zostaje zdefiniowana jako wskaźnik na te liczby (listing 3.4:1). Zmienne wskaźnikowe muszą zostać zainicjalizowane poprzez zdefiniowanie ich wskazania na poprawny obszar pamięci. W opisywanym przypadku program wykorzystuje funkcję biblioteczną `malloc` języka C w celu otrzymania z systemu adresu obszaru pamięci wystarczająco obszernego, aby przechowywać w nim liczby całkowite `c`. Argumentem funkcji `malloc` jest liczba bajtów, jakie mają zostać przydzielone. Wykonywane tu obliczenia są typowe: jest to iloczyn liczby elementów, dla których ma zostać przydzielona pamięć (`c`) oraz rozmiaru każdego elementu (`sizeof(int)`). Jeżeli pamięć systemowa zostanie wyczerpana, funkcja `malloc` wskazuje to, zwracając wartość `NULL`. Poprawnie napisane programy powinny zawsze sprawdzać wystąpienie takiej sytuacji (listing 3.4:2). Od tego miejsca program może rozwikływać wskaźnik `RRlen` (używając notacji `[]` lub operatora *), tak jakby była to tablica zawierająca elementy `c`. Należy jednak pamiętać, że nie są to metody równoważne. Funkcja `sizeof` zastosowana względem zmiennej tablicowej zwraca rozmiar tablicy (przykładowo 40 dla tablicy zawierającej 10 czterobajtowych liczb całkowitych), natomiast zastosowana względem wskaźnika zwraca jedynie wartość wymaganą do jego przechowywania w pamięci (na przykład 4 w przypadku wielu współczesnych architektur). Wreszcie, kiedy przydzielona pamięć nie jest już potrzebna, musi zostać zwolniona poprzez wywołanie funkcji `free`. Od tego momentu próba rozwikłania wskaźnika będzie prowadzić do niezdefiniowanego wyniku. Niezwolnienie pamięci jest błędem, który może prowadzić do tego, że program będzie powodował *wycieki pamięci* (ang. *memory leaks*), które powodują stopniowe marnowanie zasobów pamięciowych systemu.

Listing 3.4. *Dynamiczne przydzielanie pamięci*

```
int
update_msg(uchar *msg, int *msglen, int Vlist[], int c)
{
    [...]
    int *RRlen;                                          [1] Wskaźnik na liczbę całkowitą
    [...]                                                Liczba elementów
    RRlen = (int *)malloc((unsigned)c*sizeof(int));      Rozmiar każdego elementu
    if (!RRlen)                                          [2] Obsługa przypadku wyczerpania pamięci
        panic(errno, "malloc(RRlen)");
    [...]
    for (i = 0;  i < c;  i++) {                          Iteracyjne przejście po wszystkich elementach
        [...]

        RRlen[i] = dn_skipname(cp, msg + *msglen);       Użycie RRlen jako tablicy
        [...]
    }
    [...]
    free((char *)RRlen);                                 Zwolnienie przydzielonej pamięci
    return (n);
}
```

[43] *netbsdsrc/usr.sbin/named/named/ns_validate.c*: 871 – 1231.

Kiedy pamięć zostanie wyczerpana, program nie może zbyt wiele zrobić. W większości sytuacji jedyną skuteczną strategią postępowania jest wyświetlenie komunikatu o błędzie i wyjście. Z tego względu, oraz w celu uniknięcia konieczności sprawdzania po każdym wywołaniu wartości zwróconej przez funkcję `malloc`, może zostać ona otoczona funkcją (zwykle noszącą nazwę `xmalloc`), która wykonuje owo sprawdzenie i jest zawsze wywoływana zamiast funkcji `malloc`, jak w poniższym przykładzie[44].

```
void *
xmalloc(u_int size)
{
    void *p;

    if ((p = malloc(size)) == NULL)
        err(FATAL, "%s", strerror(errno));
    return (p);
}
    [...]
    oe = xmalloc(s);
    (void)regerror(errcode, preg, oe, s);
```

W pewnych przypadkach rozmiar tablicy jest określany w momencie, gdy są do niej wstawiane elementy. Dzieje się tak zazwyczaj wtedy, kiedy przetwarzaniu podlegają dane wejściowe: dane są przechowywane w tablicy, jednak liczba elementów, które muszą być przechowane, nie jest znana do momentu aż wszystkie one zostaną odczytane. Listing 3.5[45] ilustruje taki przypadek. Zanim element zostanie zachowany, bieżący indeks tablicy w stosunku do rozmiaru elementów w przydzielonym bloku pamięci poddawany jest sprawdzeniu. Jeżeli wymagana jest większa ilość pamięci, zostaje wywołania funkcja biblioteczna języka C `realloc` w celu dostosowania przestrzeni, na którą wskazuje pierwszy argument wywołania do nowego rozmiaru określonego przez drugi argument funkcji. Funkcja zwraca wskaźnik na dostosowany blok pamięci, ponieważ jego adres może być inny od adresu oryginalnego bloku. W takim przypadku zawartość oryginalnego bloku jest kopiowana do nowej lokacji. Należy pamiętać, że wszelkie zmienne wskaźnikowe wskazujące na lokacje należące do oryginalnego bloku będą od tego momentu wskazywać na niezdefiniowane dane. W przykładzie z listingu 3.5 rozmiar bloku pamięci jest liniowo zwiększany po 16 bajtów za każdym razem, gdy przydzielona pamięć zostanie wyczerpana. Często spotyka się również wykładniczy przyrost przydzielanej przestrzeni[46].

```
if (cur_pwtab_num + 1 > max_pwtab_num) {
    /* need more space in table */
    max_pwtab_num *= 2;
    pwtab = (uid2home_t *) xrealloc(pwtab,
                            sizeof(uid2home_t) * max_pwtab_num);
```

Listing 3.5. *Dostosowanie przydziału pamięci*

```
void
remember_rup_data(char *host, struct statstime *st)
{
        if (rup_data_idx >= rup_data_max) {      <- Czy indeks większy od przydzielonego rozmiaru?
                rup_data_max += 16;              <- Nowy rozmiar
```

[44] *netbsdsrc/usr.bin/sed/misc.c*: 63 – 107.

[45] *netbsdsrc/usr.bin/rup/rup.c*: 146 – 164.

[46] *netbsdsrc/usr.sbin/amd/hlfsd/homedir.c*: 521 – 525.

```
                rup_data = realloc (rup_data,                    ← Dostosowanie przydziału
                rup_data_max * sizeof(struct rup_data));
                if (rup_data == NULL) {
                        err (1, "realloc");
                }
        }
    rup_data[rup_data_idx].host = strdup(host);    ← Przechowanie danych
    rup_data[rup_data_idx].statstime = *st;
    rup_data_idx++;                                ← Nowy indeks
}
```

3.4.1. Zarządzanie wolną pamięcią

Powyżej wspomniano, że niezwalnianie pamięci prowadzi do sytuacji, w których programy powodują jej wyciekanie. Jednak często można spotkać takie programy, w których przydziela się, ale nie zwalnia pamięci[47]. Ich twórcy mogą sobie na to pozwolić, jeżeli działanie programów jest krótkie — w momencie zakończenia działania cała przydzielona programowi pamięć jest automatycznie przejmowana przez system operacyjny. Podobnie jest w przypadku implementacji programu *skeyinit*[48].

```
int
main(int argc, char *argv[])
{
    [...]
    skey.val = (char *)malloc(16 + 1);
    [... brak wywołania free(skey.val) ]
    exit(1)
}
```

Program *skeyinit* jest używany do zmiany hasła lub dodania użytkownika w systemie uwierzytelniania Bellcore S/Key. Wykonuje on swoje działania, a potem natychmiast kończy działanie, zwalniając zajmowaną pamięć. Jednak taka nieostrożna praktyka kodowania może powodować problemy, kiedy ten sam kod zostanie wykorzystany ponownie w programie o znacznie dłuższym czasie działania (na przykład jako część systemu oprogramowania routera). W takim przypadku program będzie powodował wycieki pamięci, co można sprawdzić korzystając z polecenia przeglądania procesów, takiego jak `ps` lub `top` w systemach uniksowych albo menedżera zadań w systemie Windows. Warto zauważyć, że w powyższym przypadku można by po prostu użyć rozwiązania, gdzie program ustawiałby wartość `skey.val` tak, aby zmienna wskazywała na tablicę o stałym rozmiarze, której pamięć przydzielono jako zmiennej lokalnej na stosie.

```
main (int argc, char *argv[])
{
    char valspace[16 + 1];
    [...]
    skey.val = valspace;
```

Często popełnianym przez początkujących programistów piszących w językach C i C++ błędem jest przyjęcie założenia, że wszystkie wskaźniki muszą zostać zainicjalizowane tak, aby wskazywały na bloki pamięci przydzielone za pomocą funkcji `malloc`. Choć kod zapisany w takim stylu nie jest błędny, program wynikowy jest często trudniej czytać i konserwować.

[47] *netbsdsrc/bin/mv/mv.c*: 260.

[48] *netbsdsrc/usr.bin/skeyinit/skeyinit.c*: 34 – 233.

W kilku przypadkach można spotkać programy, które zawierają *mechanizm przywracania pamięci* (ang. *garbage collector*), automatycznie zwalniający nieużywaną pamięć. Taka technika jest często wykorzystywana w sytuacji, gdy przydzielone bloki pamięci jest trudno śledzić, ponieważ na przykład są one współużytkowane przez różne zmienne. W takich sytuacjach z każdym blokiem zostaje związany *licznik referencji* (ang. *reference count*). Jego wartość jest zwiększana za każdym razem, gdy zostaje utworzone nowe odwołanie do bloku[49]:

```
req.ctx = ctx;
req.event.time = time;
ctx->ref_count++;
```

i zmniejszana za każdym razem, gdy referencja zostaje zniszczona[50].

```
XtFree((char*)req);
ctx->req = NULL;
ctx->ref_count--;
```

Kiedy licznik referencji osiąga wartość 0, blok nie jest dłużej używany i może zostać zwolniony[51].

```
if (--ctx->ref_count == 0 && ctx->free_when_done)
    XtFree((char*)ctx);
```

Inne, rzadziej stosowane podejście polega na użyciu *konserwatywnego mechanizmu przywracania pamięci* (ang. *conservative garbage collector*), który bada całą pamięć procesu, szukając adresów odpowiadających istniejącym, przydzielonym blokom pamięci. Wszystkie bloki, które nie zostaną znalezione w toku procesu przeglądania, są później zwalniane.

W końcu, niektóre wersje bibliotek języka C implementują niestandardową funkcję o nazwie `alloca`. Przydziela ona blok pamięci używając takiego samego interfejsu jak `malloc`, jednak zamiast przydzielać blokowi pamięć na *stercie* (ang. *heap*) programu (jest to pamięć ogólnego przeznaczenia należąca do programu) przydziela ją ona na *stosie* (ang. *stack*) programu (jest to obszar używany do przechowywania adresów zwrotnych funkcji oraz zmiennych lokalnych)[52].

```
int
ofisa_intr_get(int phandle, struct ofisa_intr_desc *descp,
               int ndescs)
{
    char *buf, *bp;
    [...]
    buf = alloca(i);
```

Blok zwrócony przez funkcję `alloca` jest automatycznie zwalniany, kiedy następuje powrót z funkcji, w której ją przydzielono. Nie ma potrzeby wywoływania funkcji `free` w celu zwolnienia przydzielonego bloku. Oczywiście, adres pamięci przydzielonej przez funkcję `alloca` nie powinien nigdy być przekazywany do podprogramów wywołujących funkcję, w której ją przydzielono, gdyż w momencie wyjścia z funkcji adres ten staje się

[49] *XFree86-3.3/xc/lib/XT/Selection.c*: 1540 – 1542.

[50] *XFree86-3.3/xc/lib/XT/Selection.c*: 744 – 746.

[51] *XFree86-3.3/xc/lib/XT/Selection.c*: 563 – 564.

[52] *netbsdsrc/sys/dev/ofisa/ofisa.c*: 225 – 244.

niepoprawny. Funkcji `alloca` nie należy używać w pewnych środowiskach programistycznych, takich jak FreeBSD, gdyż jest ona uważana za nieprzenośną i zależną od danej maszyny. W innych przypadkach, takich jak środowisko GNU, jej używanie jest zalecane, gdyż redukuje liczbę przypadkowych wycieków pamięci.

3.4.2. Struktury z dynamicznie przydzielanymi tablicami

i Niekiedy pojedyncza przydzielona dynamicznie struktura jest używana w celu przechowywania pewnych pól oraz tablicy zawierającej specyficzne dla struktury dane o zmiennej długości. Taka konstrukcja jest wykorzystywana w celu uniknięcia pośredniości wskaźników oraz narzutu pamięciowego związanego z posiadaniem przez element struktury wskaźnika na dane o zmiennej długości. Zatem zamiast definicji podobnej do poniższej[53]:

```
typedef struct {
    XID       id_base;
    [...]
    unsigned char    *data;
    unsigned long    data_len;   /* in 4-byte units */
} XRecordInterceptData;
```

użyto by podobnej do następującej[54]:

```
typedef struct {
    char *user;
    char *group;
    char *flags;
    char data[1];
} NAMES;
```

Tablica `data` — element struktury — jest używana jako miejsce na faktyczne dane. W momencie przydziału pamięci przechowującej strukturę jej rozmiar jest rozszerzany w oparciu o liczbę elementów w tablicy `data`. Od tego momentu element tablicowy jest używany tak, jakby zawierał przestrzeń dla tych elementów[55].

```
if ((np = malloc(sizeof(NAMES) +
     ulen + glen + flen + 3)) == NULL)
        err(1, "%s", "");
np->user = &np->data[0];
(void)strcpy(np->user, user);
```

Warto zauważyć, w jaki sposób w powyższym przykładzie przydzielana jest pamięć o jeden bajt większa od faktycznie potrzebnej. Rozmiar bloku pamięci jest obliczany jako suma rozmiaru struktury oraz powiązanych elementów danych: rozmiaru trzech ciągów znaków (`ulen`, `glen`, `flen`) i odpowiednich trzech znaków zerowych kończących ciągi. Jednak rozmiar struktury uwzględnia już jeden bajt dla powiązanych danych,
⚠ który nie jest brany pod uwagę w czasie obliczania rozmiaru bloku pamięci. Zarządzanie pamięcią na najniższym poziomie jest działaniem ryzykownym i podatnym na błędy.

[53] *XFree86-3.3/xc/include/extensions/record.h*: 99 – 108.

[54] *netbsdsrc/bin/ls/ls.h*: 69 – 74.

[55] *netbsdsrc/bin/ls/ls.c*: 470 – 475.

Ćwiczenie 3.11. Zlokalizuj na płycie dołączonej do książki średniej wielkości program napisany w języku C, który wykorzystuje dynamiczne przydzielanie pamięci. Oceń odsetek całości kodu programu związany z zarządzaniem dynamicznie przydzielanymi strukturami. Oszacuj wartość ponownie, zakładając, że zwalnianie pamięci jest wykonywane automatycznie przez mechanizm czyszczenia pamięci, tak jak ma to miejsce w przypadku programów pisanych w języku Java.

Ćwiczenie 3.12. Większość współczesnych środowisk programistycznych oferuje specjalizowane biblioteki, opcje kompilacji lub inne narzędzia służące do wykrywania wycieków pamięci. Zidentyfikuj mechanizmy dostępne w Twoim środowisku i użyj ich w przypadku trzech różnych programów. Przeanalizuj otrzymane wyniki.

3.5. Deklaracje typedef

Przykłady z poprzedniego podrozdziału zawierały deklaracje `typedef` służące do tworzenia nazw nowych typów danych. Deklaracja `typedef` dodaje nową nazwę (synonim) dla już istniejącego typu. Zatem po umieszczeniu poniższej deklaracji[56]:

```
typedef unsigned char cc_t;
```

widząc zapis `cc_t` należy go odczytywać jako `unsigned char`. Programy pisane w języku C używają deklaracji `typedef` w celu zapewnienia obsługi abstrakcji, zwiększenia czytelności kodu, zapobieżenia problemom związanym z przenośnością oraz emulowania mechanizmu deklaracji klas znanego z języków C++ oraz Java.

Wspólne użycie typów przedrostkowych i przyrostkowych w deklaracjach języka C czasem sprawia, że odczytanie deklaracji `typedef` staje się utrudnione[57].

```
typedef char ut_line_t[UT_LINESIZE];
```

Jednakże rozszyfrowanie takich deklaracji nie nastręcza większych problemów. Wystarczy traktować zapis `typedef` jako specyfikator przechowywania klasy, podobny do `extern` lub `static`, i odczytywać deklarację jako definicję zmiennej.

```
static char ut_line_t[UT_LINESIZE];
```

Nazwa definiowanej zmiennej (w powyższym przypadku `ut_line_t`) jest nazwą typu. Typem zmiennej jest typ odpowiadający tej nazwie.

Kiedy deklaracja `typedef` jest używana jako mechanizm abstrakcyjny, nazwa abstraktu jest definiowana jako synonim jego konkretnej implementacji. W rezultacie kod, który wykorzystuje zadeklarowaną nazwę, jest lepiej udokumentowany, gdyż jest on zapisany w kontekście odpowiednio nazwanej konstrukcji abstrakcyjnej, a nie przypadkowych szczegółów implementacji.

[56] *netbsdsrc/libexec/telnetd/defs.h*: 124.

[57] *netbsdsrc/libexec/rpc.rusersd/rusers_proc.c*: 86.

W poniższym przykładzie DBT definiuje bazę danych *thang*, strukturę zawierającą klucz lub element danych[58].

```
typedef struct {
    void   *data;         /* data */
    size_t  size;         /* data length */
} DBT;
```

Po tej deklaracji wszystkie podprogramy obsługi dostępu do bazy danych są definiowane jako działające na obiektach DBT (i innych zdefiniowanych za pomocą deklaracji typedef)[59].

```
int    __rec_get (const DB *, const DBT *, DBT *, u_int);
int    __rec_iput (BTREE *, recno_t, const DBT *, u_int);
int    __rec_put (const DB *dbp, DBT *, const DBT *, u_int);
int    __rec_ret (BTREE *, EPG *, recno_t, DBT *, DBT *);
```

Ze względu na fakt, że w przypadku języków C i C++ szczegóły sprzętowe typów danych języka zależą od odpowiedniej architektury, kompilatora oraz systemu operacyjnego,

i deklaracje typedef są często używane w celu zwiększenia przenośności programu albo poprzez utworzenie przenośnych nazw dla znanych wielkości sprzętowych, albo przez szereg deklaracji zależnych od implementacji[60]

```
typedef   __signed char     int8_t;
typedef   unsigned char   u_int8_t;
typedef   short             int16_t;
typedef   unsigned short  u_int16_t;
typedef   int               int32_t;
typedef   unsigned int    u_int32_t;
typedef   long              int64_t;
typedef   unsigned long   u_int64_t;
```

i albo tworząc abstrakcyjne nazwy dla wielkości o znanej reprezentacji sprzętowej, przy użyciu jednej z wcześniej zadeklarowanych nazw[61].

```
typedef u_int32_t in_addr_t;
typedef u_int16_t in_port_t;
```

i Wreszcie deklaracje typedef są również często używane w celu emulowania znanego z języków C++ i Java mechanizmu, kiedy to deklaracja klasy wprowadza nowy typ. W programach pisanych w C często spotyka się deklarację typedef użytą w celu wprowadzenia nazwy typu dla struktury (jest to najbliższa klasie konstrukcja występująca w C) identyfikowanej przez tę samą nazwę. Zatem poniższy przykład deklaruje path jako synonim dla struct path[62].

```
typedef struct path path;
struct path {
    [...]
```

[58] *netbsdsrc/include/db.h*: 72 – 75.

[59] *netbsdsrc/lib/libc/db/recno/extern.h*: 47 – 50.

[60] *netbsdsrc/sys/arch/alpha/include/types.h*: 61 – 68.

[61] *netbsdsrc/sys/arch/arm32/include/endian.h*: 61 – 62.

[62] *netbsdsrc/sbin/mount_portal/conf.c*: 62 – 63.

Ćwiczenie 3.13. Zlokalizuj na płycie dołączonej do książki pięć różnych wystąpień każdego rodzaju użycia deklaracji `typedef`, o których była mowa.

Ćwiczenie 3.14. W jaki sposób deklaracje `typedef` mogą negatywnie wpływać na czytelność kodu?

Dalsza lektura

Jeżeli zagadnienia omówione w niniejszym rozdziale nie są znane Czytelnikowi, warto poszerzyć swoją znajomość języka C dzięki lekturze pozycji [KR88]. Teoretyczne podstawy implementacji rekurencyjnych typów danych bez jawnego użycia wskaźników przedstawiono w pozycji Hoare'ego [Hoa73]. Użycie wskaźników w języku C zostało zwięźle przedstawione w pozycji Sethiego i Stone'a [SS96], zaś wiele pułapek związanych z ich użyciem omówiono w pozycji Koeniga [Koe88, s. 27 – 46]. W pozycji [CWZ90] zawarto analizę wskaźników i struktur. Istnieje wiele interesujących artykułów poświęconych manipulowaniu strukturami danych opartymi na wskaźnikach i wykorzystywaniu właściwości wskaźników [FH82, Suz82, LH86]. Opis sposobu realizacji funkcji wirtualnych w implementacjach języka C++ (jak również, zgodnie z informacjami podanymi w niniejszym rozdziale, w języku C) można znaleźć w pozycji Ellis i Stroustrupa [ES90, s. 217 – 237]. Algorytmy związane z dynamicznym przydzielaniem pamięci omówiono w pozycji Knutha [Knu97, s. 435 – 452], zaś związane z nimi praktyczne implikacje w dwóch innych źródłach [Bru82, DDZ94]. Pojęcie mechanizmu oczyszczania pamięci z licznikiem referencji omówiono w pozycji Christophera [Chr84], natomiast zarys implementacji konserwatywnego mechanizmu oczyszczania pamięci można znaleźć w pozycji Boehma [Boe88].