Programowanie w elektronice: Podstawy C i C++

Projekt „Matematyka dla Ciekawych Świata”,
Robert Ryszard Paciorek
<rrp@opcode.eu.org>

2024-10-03

C / C++ są najpopularniejszymi językami kompilowanymi do kodu maszynowego (a jeżeli traktować je łącznie to najpopularniejszymi językami w ogóle), pozwalają na stosowanie niskopoziomowych mechanizmów (łącznie z wstawkami asemblerowymi), są użyteczne do bezpośredniego programowania sprzętu (bez warstwy systemu operacyjnego) czy też tworzenia systemów operacyjnych.

Język C jest językiem kompilowalnym to znaczy (po zmodyfikowaniu źródeł) przed uruchomieniem programu konieczne jest dokonanie tłumaczenia kodu źródłowego na kod maszynowy przy pomocy odpowiedniego programu (np. clang lub gcc). Kompilacja przebiega kilku etapowo. W pierwszej kolejności wywoływany jest preprocesor, który jest odpowiedzialny za włączanie plików określonych poprzez `#include` (jest to literalne włączenie zawartości wskazanego pliku w danym miejscu, obsługę rozwijania stałych makr preprocesora (definiowanych z użyciem `#define`) oraz kompilację warunkową z wykorzystaniem poleceń takich jak `#ifdef` czy `#if`. Kompilatory pozwalają na uzyskanie nie tylko wynikowego pliku binarnego, ale także plików po przetworzeniu przez preprocesor czy też po konwersji na assembler.

Część poniższego kodu zakłada że używany jest C w wersji co najmniej 99, zatem do jego kompilacji powinno być użyte np. polecenie `gcc -std=c99 plik.c` (lub `clang -std=c99 plik.c`), które utworzy plik wykonywalny `a.out` (można go uruchomić poprzez `./a.out`).

1 Zmienne i podstawowe operacje

Język C wymaga określania typu zmiennej w momencie jej definiowania.

```
// liczba całkowita ze znakiem
int     liczbaA = -34;
// liczba rzeczywista (pojedynczej precyzji)
float   liczbaB = 673.1;
// 8 bitowa liczba całkowita bez znaku, wymaga pliku nagłówkowego inttypes.h
uint8_t liczbaC = 0xf3;

// zmienna napisowa "C NULL-end string"
char* napisA = "q we";
```

Dodawanie, mnożenie, odejmowanie zapisuje się i działają one tak jak w normalnej matematyce, dzielenie zapisuje się przy pomocy ukośnika i zależnie od typów na których operuje jest ono dzieleniem całkowitym lub zmiennoprzecinkowym.

```
double a = 12.7, b = 3, c, d, e;
int x = 5, y = 6, z;

// dodawanie, mnożenie, odejmowanie zapisuje się
// i działają one tak jak w normalnej matematyce:
e = (a + b) * 4 - y;

// dzielenie zależy od typów argumentów
```

```
d = a / b; // będzie dzieleniem zmiennoprzecinkowym bo a i b są typu float
c = x / y; // będzie dzieleniem całkowitym bo z i y są zmiennymi typu int
b = (int)a / (int)b; // będzie dzieleniem całkowitym
a = (double)x / (double)y; // będzie dzieleniem zmiennoprzecinkowym

// reszta z dzielenia (tylko dla argumentów całkowitych)
z = x % y;

// wypisanie wyników
printf("%d %f %f %f %f %f\n", z, e, d, c, b, a);

// operacje logiczne:
// ((a większe równe od 0) AND (b mniejsze od 2)) OR (z równe 5)
z = (a>=0 && b<2) || z == 5;
// negacja logiczna z
x = !z;

printf("%d %d\n", z, x);

// operacje binarne:
// bitowy OR 0x0f z 0x11 i przesunięcie wyniku o 1 w lewo
x = (0x0f | 0x11) << 1;
// bitowy XOR 0x0f z 0x11
y = (0x0f ^ 0x11);
// negacja bitowa wyniku bitowego AND 0xfff i 0x0f0
z = ~(0xfff & 0x0f0);

printf("%x %x %x\n", x, y, z);
```

printf() i puts()

Funkcja `printf()` wypisuje napis określony przez pierwszy argument, podstawiają pod elementy typu %x wartości kolejnych argumentów (np. zmiennych) odpowiednio je interpretując, np.:

- %x - liczba dziesiętna,
- %x - liczba szesnastkowa,
- %f - liczba zmiennoprzecinkowa,
- %s - napis,
- %c - pojedynczy znak.

Funkcja ta nie dodaje do wypisywanego napisu znaku nowej linii, więc jeżeli chcemy przejść do nowej linii to musi on być umieszczony w nim w sposób jawny jako \n (niekiedy jako \r\n - protokoły sieciowe, Windows, ...). Szczegółowy opis `printf()` oraz więcej napisów formatujących można znaleźć w `man 3 printf`.

Do wypisywania samego napisu (bez możliwości podstawienia zmiennych, itd.) posłużyć może funkcja `puts()`, która wypisuje podany napis dodając znak nowej linii.

2 Przepływ sterowania w programie - skoki, warunki, pętle, funkcje

Licznik programu (program counter, instruction pointer lub instruction address register) jest rejestrem procesora który określa adres następnej (w niektórych architekturach aktualnej) instrukcji która ma zostać przetworzona procesor.

Skoki bezwarunkowe, instrukcje warunkowe, pętle, wywołania funkcji są realizowane poprzez modyfikację licznika programu. W przypadku wywołań funkcji dodatkowo wykonywane są operacje związane z obsługą stosu (zachowywaniem stanu rejestrów, umieszczaniem argumentów na stosie, ...). Instrukcja goto (realizująca skok bezwarunkowy) jest pełnoprawną instrukcją skoku, jedyną wadą jej stosowania jest to że przy niewłaściwym / zbyt częstym wykorzystywaniu (zamiast wywołań funkcji, warunków i pętli) kod programu staje się mniej czytelny.

W większości przypadków pętle realizowane są na poziomie kodu maszynowego jako zestaw instrukcji (np. inkrementacji zmiennej, sprawdzania warunku, skoku), jednak w niektórych rozwiązaniach pętle (np. typu "powtórz n razy") mogą być realizowane sprzętowo przy pomocy pojedynczej instrukcji.

2.1 Punkt startu

Jako że program komputerowy jest sekwencją wykonywanych instrukcji musi rozpoczynać się od określonego miejsca. W przypadku kodu C/C++ punktem startu jest funkcja `main()`. Zakończenie tej funkcji oznacza zakończenie programu, a wartość przez nią zwracana odpowiedzialna jest za tzw. kod powrotu przekazany procesowi wywołującemu program.

Zadanie 2.1.1

Napisz program wypisujący "Hello World".

2.2 Podstawowe konstrukcje

Język C oferuje kilka pętli – **`for`**, **`while`**, **`do - while`** oraz instrukcję warunkową **`if`** i instrukcję wyboru **`switch`**. Możliwe jest też korzystanie z operatora warunkowego:

```
warunek ? wartosc_gdy_prawda : wartosc_gdy_falsz.
```

Gdzie zarówno `wartosc_gdy_prawda`, jak i `wartosc_gdy_falsz`, mogą być wartością jak też wyrażeniem obliczającym jakąś wartość (wyrażeniem matematycznym, wywołaniem funkcji, itd.).

```c
#include <stdio.h> // włączenie pliku nagłówkowego

int main() {
    int i, j, k;

    // instrukcja waunkowa if - else
    if (i<j) {
        puts("i<j");
    } else if (j<k) {
        puts("i>=j AND j<k");
    } else {
        puts("i>=j AND j>=k");
    }

    // podstawowe operatory logiczne
    if (i<j || j<k)
        puts("i<j OR j<k");
    // innymi operatorami logicznymi są && (AND), ! (NOT)

    // pętla for
    for (i=2; i<=9; ++i) {
        if (i==3)
```

```
            continue; // pominięcie tego kroku pętli
        if (i==7)
            break; // wyjście z pętli
        printf(" a: %d\n", i);
    }

    // pętla while
    while (i>0) {
        printf(" b: %d\n", --i);
    }

    // pętla do - while
    do {
        printf(" c: %d\n", ++i);
    } while (i<2);

    // instrukcja wyboru switch
    switch(i) {
        case 1:
            puts("i==1");
            break;
        default:
            puts("i!=1");
            break;
    }
}
```

Zadanie 2.2.1

Zmodyfikuj program z zadania 2.1.1 tak aby z użyciem pętli wypisywał ten napis 10 razy.

2.3 Własne funkcje

```
#include <stdio.h>

// funkcja bezargumentowa niezwracająca wartości
void f1() {
    puts("ABC");
}

// funkcja dwuargumentowa zwracająca wartość
int f2(int a, int b) {
    puts("F2");
    return a*2.5 + b;
}

int main() {
    f1();
    int a = f2(3, 6);
    // zwracaną wartość można wykorzystać (jak wyżej) lub zignorować
    printf("%d\n", a);
```

```
}
```

3 Złożone typy danych

3.1 Struktury

Struktura jest złożonym typem danych służącym do grupowania powiązanych ze sobą logicznie zmiennych. Zmienne wchodzące w skład struktury (pola) identyfikowane są nazwami i mogą być różnych typów. Struktura zajmuje ciągły obszar pamięci, w którym umieszczane są wartości kolejnych pól.

```c
#include <stdio.h>

struct Struktura {
    int a, b;
    double c;
};

int main() {
    struct Struktura s;
    s.a = 13;
    s.c = 17.3;
    printf("%f\n", s.a + s.c);
}
```

3.2 Tablice

Tablica jest strukturą danych w której elementy (takiego samego typu) są ułożone w porządku liniowym i są dostępne za pomocą indeksów (kluczy). Typowo tablica indeksowana jest liczbami całkowitymi nie ujemnymi oraz zajmuje ciągły obszar pamięci.

```c
#include <stdio.h>

int main() {
    int t[4] = {1, 8, 3, 2};
    printf("%d -> \n", t[2]);
    t[2] = 55;
    printf("    %d \n", t[2]);
}
```

3.3 Napisy

Napisy w C są w istocie tablicami elementów typu `char`. Pojedynczy znak reprezentowany jest poprzez jeden element tablicy (dla znaków kodowanych jednobajtowo) lub grupę takich elementów (dla znaków kodowanych wielobajtowo, np. polskich znaczków w UTF8). Koniec napisu oznaczany jest przez element o wartości zero (`NULL`).

W C pojedynczy znak napisu (czyli np. `char x = napis[i]` albo `char x = 'A';`) nie jest napisem – jest liczbą (zauważ różnicę między pojedynczymi a podwójnymi cudzysłowami). Możemy go wypisać z użyciem `printf()` jako wartość numeryczną poprzez `%d` lub jako znak poprzez `%c`: `printf("%d <=> %c", 'A', 'A');`.

Zadanie 3.3.1

Napisz funkcję `wypisz`, która z użyciem pętli będzie wypisywała poszczególne znaki podanego napisu od wskazanej do wskazanej pozycji. Przyjmij że napis składa się tylko z znaków ASCII.
Wywołanie `wypisz("! Hello World !", 3, 6);` powinno spowodować wypisanie `ello`.

Zadanie 3.3.2

Zmodyfikuj program z zadania 3.3.1 tak aby funkcja, w przypadku gdy argument określający ostatni znak podnapisu do wypisania jest większy niż długość napisu, kończyła swoje działanie w momencie napotkania końca napisu.
Wywołanie `wypisz("! Hello World !", 10, 116);` powinno spowodować wypisanie `rld !` i niczego więcej.

Zadanie 3.3.3

Zmodyfikuj rozwiązanie zadania 3.3.2 tak aby poprawnie obsługiwało znaki kodowane jako UTF8.
Wskazówka 1: Zobacz opis kodowania UTF-8 na https://en.wikipedia.org/wiki/UTF-8, zauważ że w bajtach stanowiących kontynuację znaku pierwsze dwa bity mają wartość 10, natomiast pierwszy bajt znaku nigdy nie ma takiej wartości najstarszych bitów.
Wskazówka 2: Zauważ, że aby odnaleźć pierwszy znak do wypisania, musisz przejść po napisie od samego początku.

4 Funkcja main

Jak już zauważyliśmy funkcja `main()` zwraca wartość całkowitą. Jest to kod powrotu programu, który służy do informowania procesu wywołującego nasz program o tym czy zakończył się on sukcesem czy porażką. W przypadku sukcesu powinien zwrócić 0, a niezerowa wartość oznacza niepowodzenie (można użyć różnych wartości do sygnalizowania różnego rodzaju niepowodzeń - np. polecenie grep inaczej sygnalizuje nie znalezienie podanego wzorca, a inaczej brak pliku który miało przeszukać).

Funkcja `main()` może przyjmować także argumenty, dzięki którym program może odebrać parametry przekazane w linii poleceń. Drugi argument zazwyczaj nazywany *argv* jest tablicą napisów **char** *, której elementy stanowią kolejne słowa (ciągi znaków oddzielane niezabezpieczonymi spacjami) składające się na linię polecenia w wyniku którego został uruchomiony program (czyli nazwa polecenia, opcje i argumenty)

Pierwszy argument jest liczbą całkowita typu **int** i określa ilość elementów tablicy przekazanej jako drugi argument.

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
        for (int i=0; i<argc; ++i)
                printf("element %d to: %s\n", i, argv[i]);

        return argc - 2; // kod powrotu uzależniamy od ilości argumentów
}
```

Przykład działania (linie zaczynające się od $ są wprowadzonymi komendami, pozostałe linie to output uruchomionych poleceń):

```
$ ./a.out a b; echo "Kod powrotu $?"
element 0 to: ./a.out
element 1 to: a
element 2 to: b
```

```
Kod powrotu 1
$ ./a.out a; echo "Kod powrotu $?"
element 0 to: ./a.out
element 1 to: a
Kod powrotu 0
```

Zauważ że tablica otrzymana jako drugi argumnent zawsze ma co najmniej jeden element - nazwę uruchomionego polecenia.

Porada

Argumenty linii poleceń są na ogół dużo lepszą metodą odbierania danych od użytkownika niż zadawanie pytań i oczekiwanie na wprowadzenie danych z klawiatury na standardowe wejście – umożliwiają łatwe wykorzystanie naszych programów w bardziej złożonych poleceniach, czy skryptach. Zastanów się jakby wyglądało programowanie w bashu gdyby komendy takie jak ls, grep, find, itd oczekiwały na wprowadzenie swoich opcji i argumentów na standardowym wejściu.

5 Zmienna i jej adres

Wszelkie dane na których operuje program komputerowy przechowywane są w jakimś rodzaju pamięci - najczęściej jest to pamięć operacyjna. W pewnych sytuacjach niektóre dane mogą być przechowywane np. tylko w rejestrach procesora lub rejestrach urządzeń wejścia-wyjścia.

W programowaniu na poziomie wyższym od kodu maszynowego i asemblera używa się pojęcia zmiennej i (niemal zawsze) pozostawia kompilatorowi/interpretatorowi decyzję o tym gdzie ona jest przechowywana. Oczywistym wyjątkiem są grupy zmiennych, czy też bufory alokowane w sposób jawny w pamięci. Ze względu na ograniczoną liczbę rejestrów procesora większość zmiennych (w szczególności tych dłużej istniejących i większych) będzie znajdowała się w pamięci i będą przenoszone do rejestrów celem wykonania jakiś operacji na nich po czym wynik będzie przenoszony do pamięci.

Z każdą zmienną przechowywaną w pamięci związany jest *adres pamięci* pod którym się ona znajduje. Niektóre z języków programowania pozwalają na odwoływanie się do niego poprzez wskaźnik na zmienną lub referencję do zmiennej (odwołania do adresu zmiennej mogą wymusić umieszczenie jej w pamięci nawet gdyby normalnie znajdowała się tylko w rejestrze procesora).

Wszystkie dane są zapisywane w postaci liczb lub ciągów liczb. Typ zmiennej (jawny lub nie) informuje o tym jakiej długości jest dana liczba i jak należy ją interpretować (jak należy interpretować ciąg liczb).

5.1 Zasięg zmiennej

Zasięg zmiennych (widoczność i istnienie) jest limitowany do bloku (wydzielanego nawiasami klamrowymi) w którym zostały zadeklarowane, zmienne z bloków wewnętrznych mogą przesłaniać zmienne zadeklarowane wcześniej.

Wywołanie funkcji powoduje rozpoczęcie nowego kontekstu w którym zmienne z bloku wywołującego funkcję nie są widoczne (ale nadal istnieją). Argumenty do funkcji przekazywane są przez kopiowanie, więc funkcja nie ma możliwości modyfikacji zmiennych z bloku ją wywołującego nawet do niej przekazanych (wyjątkiem jest przekazanie przez referencję lub wskaźnik).

W przypadku manualnej alokacji pamięci (z użyciem malloc) limitowana jest widoczność i istnienie otrzymanego wskaźnika, ale nie zaalokowanego bloku pamięci. Zatem ograniczona jest widoczność takich zmiennych ale nie czas ich istnienia, dlatego też przed utratą wskaźnika na nie należy je usunąć (zwolnić zaalokowaną pamięć).

5.2 Wskaźniki

Wskaźnik jest zmienną, która przechowuje adres pamięci, pod którym znajdują się jakieś dane (inna zmienna). Jako że wskaźnik jest zmienną która też jest umieszczona gdzieś w pamięci można utworzyć

wskaźnik do wskaźnika itd. Na wskaźnikach można wykonywać operacje arytmetyczne (najczęściej jest to dodawanie offsetu). Na wskaźniku można wykonać operację wyłuskania czyli odwołania się do wartości zmiennej pod adresem na który wskazuje, a nie do zmiennej wskaźnikowej (zawierającej adres).

Wskaźniki pozwalają na operowanie dużymi zbiorami danych (duże struktury, napisy, etc) bez konieczności ich kopiowania przy przekazywaniu do funkcji, umieszczaniu w różnych strukturach danych, sortowaniu, itd (kopiowaniu ulega jedynie wskaźnik czyli adres) oraz na współdzielenie tych samych danych pomiędzy różnymi obiektami.

Wskaźnik może wskazywać na niewłaściwy adres w pamięci (np. na skutek zwolnienia tego fragmentu lub błędu w operacjach matematycznych na wskaźnikach - wyjściu poza dozwolony zakres), typowo wskaźnikowi który nic nie wskazuje przypisuje się wartość `NULL` (zero). Wyłuskania wskaźników o wartości `NULL` lub wskazujących niewłaściwy obszar pamięci prowadzą do błędów programu, często do zakończenia programu z powodu naruszenia ochrony pamięci ("Segmentation fault").

```c
#include <stdio.h>

int main() {
    int zm = 13;
    int *wsk = NULL; // zmienna wskaźnikowa (na typ int)

    // przypisanie do zmiennej wskaźnikowej adresu zmiennej zm
    // pobranie adresu zmiennej przy pomocy operatora &
    wsk = &zm;
    printf("%p %p\n", &zm, wsk);

    // odwołanie do zmiennej wskazywanej przez wskaźnik (wyłuskanie wartości)
    // przy pomocy operatora *
    printf("%d %d\n", zm, *wsk);
    *wsk = 17;
    printf("%d %d\n", zm, *wsk);
}
```

5.3 Wskaźniki a tablice

Zmienna tablicowa w C to w istocie wskaźnik na pierwszy element tablicy. Dostęp do elementów tablicy odbywa się w oparciu o obliczanie ich adresu na podstawie zależności: AdresElementu = AdresPoczatkuTablicy + IndexElementu * RozmiarElementu.

```c
#include <stdio.h>

int main() {
  int t[4] = {1, 8, 3, 2};
  int *tt = t; // zauważ brak operatora pobrania adresu

  printf("%d %d\n", t[2], tt[2]);
  printf("%d %d\n", *(t + 2), *(tt + 2));
}
```

Zauważ że operator `t[x]` działa tak samo dla tablicy jak i dla wskaźnika i jest w istocie ładniejszym zapisem operacji `*(t+x)` na samym wskaźniku.

5.4 Wskaźniki a funkcje

Argumenty do funkcji przekazywane są przez kopiowanie, w związku z tym modyfikacja zmiennej będącej argumentem funkcji wewnątrz tej funkcji nie będzie widoczna poza nią:

```
void ff(int a) {
    a = 15;
}
int main() {
    int x = 10;
    ff(x);
    printf("%d\n", x); // wypisze 10
}
```

Jeżeli chcemy mieć możliwość modyfikacji zmiennej przekazywanej przez argument możemy przekazać zmienną do funkcji przez wskaźnik:

```
void ff(int* a) {
    *a = 15;
}
int main() {
    int x = 10;
    ff(&x);
    printf("%d\n", x); // wypisze 15
}
```

Z rozwiązania takiego korzystamy też gdy chcemy uniknąć kopiowania dużych struktur, w tym przypadku dobrym zwyczajem jest dodanie const, aby funkcja nie mogła modyfikować tego na co wskazuje ten wskaźnik:

```
struct Struktura {
  int a, b;
};
void ff(const struct Struktura *s) {
    s->a = 15; // błąd kompilacji w tym miejscu, z powodu const w linii wyżej
    /* zauważ że do elementów struktur możemy się odwoływać
       obiekt.pole lub (&obiekt)->pole (czyli wskazik_na_obiekt->pole) */
}
int main() {
    struct Struktura s;
    ff (&s);
}
```

5.5 Arytmetyka wskaźnikowa

Jak już zauważyliśmy na wskaźnikach można wykonywać (niektóre) operacje arytmetyczne. Ich działanie jest zależne od typu wskaźnika, tj. zwiększenie wskaźnika o 1 zwiększa adres na który on wskazuje o tyle bajtów ile zajmuje zmienna której typu jest wskaźnik.

```
#include <stdio.h>

int main() {
    char a;    int    b;
```

```c
    char *wsk_a = &a;
    int    *wsk_b = &b;

    printf("char: %p %p\n", wsk_a, wsk_a+1);
    printf("int:    %p %p\n", wsk_b, wsk_b+1);
}
```

Zadanie 5.5.1

Zmodyfikuj program z zadania 3.3.2 tak aby funkcja korzystała z arytmetyki wskaźnikowej, zamiast iteracji po numerze znaku w napisie (pętla ma korzystać z wskaźników i ich porównywania, zamiast operowania na numerze/indeksie znaku!).

5.6 Kolejność bajtów 🤓

Wskaźniki i rzutowanie typów pozwala patrzeć na dane w postaci poszczególnych bajtów.

```c
#include <inttypes.h>
#include <stdio.h>
int main() {
    // dane jako tablica liczb 16 bitowych
    uint16_t aa[4] = {0x1234, 0x5678, 0x9abc, 0xdeff};

    // wypisujemy ją
    printf("A0: %x %x %x %x\n", aa[0], aa[1], aa[2], aa[3]);
    // chyba nikogo nie zaskoczy wynik powyższego printf:
    //   A0: 1234 5678 9abc deff

    // wypisujemy dwie pierwsze liczby rozłożone na części 8 bitowe
    // (poszczególne bajty)
    printf(
        "A1: %x %x %x %x\n",
        (aa[0] >> 8) & 0xff, aa[0] & 0xff,
        (aa[1] >> 8) & 0xff, aa[1] & 0xff
    );
    // efekt też jest oczywisty:  A1: 12 34 56 78

    // każemy na te same dane patrzeć jako na liczby 8 bitowe
    // (poszczególne bajty)
    uint8_t* bb = (uint8_t*) aa;

    printf("B0: %x %x %x %x\n", bb[0], bb[1], bb[2], bb[3]);
    // czego się teraz spodziewamy?
    //  - wypisze nam tylko połowę oryginalnej tablicy
    //  - ale dokładny wynik zależy od architektury na której uruchamiamy
    //    program (big endian vs little endian)
}
```

Kod ten w zależności od architektury procesora na którym będzie uruchomiony może wypisać inny wynik:

na *little endian* (np. x86):

```
A0: 1234 5678 9abc deff
A1: 12 34 12 34
B0: 34 12 78 56
```

na *big endian* (np. sparc) – zapis w "ludzkiej" kolejności:

```
A0: 1234 5678 9abc deff
A1: 12 34 12 34
B0: 12 34 56 78
```

Fakt, że różne komputery ten sam ciąg zero-jedynkowy mogą interpretować jako różne liczby (w zależności od architektury „big endian" vs „little endian"), powoduje że przy wymianie danych między systemami konieczne jest ustalenie sposobu tej interpretacji (np. protokoły sieciowe takie jak IP używają „big endian") lub zawarcie tej informacji w wymienianych danych (kodowania Unicode UTF-16 i UTF-32 zawierają na początku danych znacznik BOM).

6 C++

Jak być może zauważyliście programowanie w C jest dość mocno niżej poziomowe od programowania np. w Pythonie. A co za tym idzie często trudniejsze i wymagające więcej wysiłku i czasu. Natomiast wykonanie takiego skompilowanego do kodu maszynowego programu jest znacznie szybsze. Jeżeli w jakimś zastosowaniu potrzebujemy kompilowalnego, szybkiego języka takiego jak C, ale z wyżej poziomowymi mechanizmami, z jakimi można spotkać się na przykład w Pythonie warto zwrócić uwagę na C++. Język ten oferuje między innymi:

- dynamiczne typowanie podobne do znanego z Pythona z użycie słowa kluczowego `auto`,
- programowanie generyczne (niezależne od typów) z użyciem szablonów,
- pętlę for iterującą po elementach kolekcji, wsparcie dla programowania obiektowego, funkcji typu lambda.
- listy i słowniki i to znacznie bardziej rozbudowane niż te z którymi mieliśmy do czynienia w Pythonie, możemy mieć np. słownik z wieloma jednakowymi kluczami, zbiór samych unikalnych kluczy, itd.,

Język C powstał z rozszerzenia języka B w 1972 roku, natomiast C++ jest zapoczątkowanym w 1979 roku rozszerzeniem języka C. Pierwszy oficjalny standard C pochodzi z 1989 roku (*ANSI X3.159-1989 / ISO/IEC 9899:1990*)[1], a C++ z 1998 roku (*ISO/IEC 14882-1998*). Języki te od czasu swojego powstania, a następnie ustandaryzowania rozwijają się niezależnie. Obecnie, w 2020 roku są to dwa niezależne języki (C++17 z 2017 roku i C18 z 2018 roku), jednak cały czas bardzo bliskie sobie – jednym z założeń C++ jest zachowanie maksymalnej kompatybilności z C[2]. C++ to nie tylko „C z klasami", język ten oferuje wiele usprawnień w stosunku co do C (często adaptowanych do C w kolejnych wersjach).

Część poniższego kodu zakłada że używany jest C++ w wersji co najmniej 11, zatem do jego kompilacji powinno być użyte np. polecenie `g++ -std=c++11 plik.cpp` (lub `clang++ -std=c++11 plik.cpp`), które utworzy plik wykonywalny `a.out` (można go uruchomić poprzez `./a.out`).

6.1 programowanie obiektowe

```cpp
#include <iostream>
#include <stdint.h>

struct NazwaStruktury {
    // pola składowe
    int a;
```

1. Przy opracowywaniu tego standardu od języka C oddzieliła się część opisująca zagadnienia specyficzne dla środowisk unixowych w postaci standardu POSIX (*Portable Operating System Interface*) opisywanego w serii standardów IEEE Std 1003, publikowanych od 1988 roku.
2. Większość różnic jest do tego stopnia pomijalna (zwłaszcza biorąc pod uwagę pewną liberalność kompilatorów), że typowy program C, można skompilować jako kod C++ i będzie on poprawnie działającym programem. Często nawet jest w 100% formalnie poprawnym kodem C++ (ale niekoniecznie w stylu tego języka).

```
    std::string d;

    // zmienna statyczna, wspólna dla wszystkich obiektów tej klasy
    static int x;

    // stała
    static const int y = 7;

    // pola binarne (jedno i trzy bitowe)
    uint8_t mA :1;
    uint8_t mB :3;

    // metody składowe
    void wypisz() {
        std::cout << " a=" << a << " d=" << d << "\n";
    }

    // deklaracja metody, definicja musi być podana gdzieś indziej
    int getSum(int b) ;

    /// metody statyczna
    static void info() {
        std::cout << "INFO\n";
    }

    // konstruktor i destruktor
    NazwaStruktury(int aa = 0) {
        std::cout << "konstruktor\n";
        a = aa;
        d = "abc ...";
    }
    ~NazwaStruktury() {
        // potrzebny gdy klasa tworzy jakieś
        // obiekty które nalezy usuwać, itp
        std::cout << "destruktor\n";
    }
};

// definicja zmiennej statycznej z nadaniem jej wartości
// jest to niezbędne aby była ona widoczna ...
int NazwaStruktury::x = 13;

// wcześniej zdeklarowane metody możemy definiować także poza deklaracją klasy
int NazwaStruktury::getSum(int b) {
    return a + b;
}

int main() {
    // korzystanie ze struktur
    NazwaStruktury s;
    s.a = 45;
    s.wypisz();
```

```
    // korzystanie z metod statycznych
    NazwaStruktury::info();
    // a także poprzez obiekt danej klasy
    s.info();
}
```

6.2 tablice zmiennej długości

Język C od wersji C99 pozwala na korzystanie z tablic zmiennej długości (*VLA*), czyli tablic których rozmiar nie jest stałą czasu kompilowania a zmienną - np.:

```
void xxx(int n) {
    float vals[n];
    v[0] = 21;
    /* ... */
}
```

C++ tablic zmiennej długości w stylu C99 C++ oficjalnie nie obsługuje, przy czym niektóre z kompilatorów dopuszczają użycie VLA w C++. C++ posiada za to typ std:vector pozwalający na definiowanie tablic, których rozmiar można łatwo (z punktu widzenia programisty, niekoniecznie maszyny wykonującej ten kod) zmieniać nawet po utworzeniu tablicy:

```
void xxx(int n) {
    std::vector<float> vals(n);
    v[0] = 21;
    /* ... */
}
```

6.3 listy

Biblioteka standardowa C++ (a dokładniej jej fragment określany mianem STL) dostarcza także obsługę list:

```
#include <iostream>
#include <list>

int main() {
    std::list<int> l;

    // dodanie elementu na końcu
    l.push_back(17);
    l.push_back(13);
    l.push_back(3);
    l.push_back(27);
    l.push_back(21);
    // dodanie elementu na początku
    l.push_front(8);

    // wypisanie liczby elementów
    std::cout << "size=" << l.size()<< "\n";
```

```
    // wypisanie pierwszego i ostatniego elementu
    std::cout << "first=" << l.front() << " last=" << l.back() << "\n";

    // usuniecie ostatniego elementu
    l.pop_back();

    // posortowanie listy
    l.sort();

    // odwrócenie kolejności elementów
    l.reverse();

    // usuniecie pierwszego elementu
    l.pop_front();

    for (std::list<int>::iterator i = l.begin(); i != l.end(); ++i) {
        // wypisanie wszystkich elementów
        std::cout << *i << "\n";
        // możliwe jest także:
        //  - usuwanie elementu wskazanego przez iterator
        //  - wstawianie elementu przed wskazanym przez iterator
    }
}
```

W przypadku C++ listy implementowane są jako listy a nie tablice wskaźników, więc operacje wstawiania na początku i w środku są szybkie, ale operacja uzyskania n-tego elementu jest powolna.

6.4 mapy

Biblioteka standardowa C++ oferuje także kontener umożliwiający przechowywanie danych w postaci par klucz-wartość, gdzie wartość identyfikowana jest unikalnym kluczem (podobnie jak w pythonowych słownikach):

```
#include <iostream>
#include <map>

int main() {
    std::map<std::string, int> m;

    m["a"] = 6;
    m["cd"] = 9;
    std::cout << m["a"] << " " << m["ab"] << "\n";

    // wyszukanie elementu po kluczu
    std::map<std::string, int>::iterator iter = m.find("cd");
    // sprawdzenie czy istnieje
    if (iter != m.end()) {
        // wypisanie pary - klucz wartość
        std::cout << iter->first << " => " << iter->second << "\n";
        // usunięcie elementu
        m.erase(iter);
    }
```

```
    m["a"] = 45;

    // wypisanie całej mapy
    for (iter = m.begin(); iter != m.end(); ++iter)
        std::cout << iter->first << " => " << iter->second << "\n";
    // jak widać mapa jest wewnętrznie posortowana
}
```

Mapa `std::map` nie zachowuje kolejności wkładania elementów, natomiast jest zawsze posortowana. C++ oferuje też inne rodzaje map (np. nie posortowaną `std::unordered_map`, czy też nie wymagającą unikalności klucza `std::multimap`).

Zadanie 6.4.1

Napisz funkcję która konwertuje listę napisów postaci `klucz=wartosc` na mapę. Funkcja musi dodawać kolejne napisy do mapy w taki sposób że część przed znakiem równości stanowi klucz, a część po znaku równości stanowi wartość. Funkcja powinna modyfikować mapę otrzymaną (przez wskaźnik lub referencję) jako swój argument. Na przykład dla wywołania:

```
std::list<std::string> l = {"aa=13", "b=Ala=kot", "f=xyz"};
std::map<std::string,std::string> m;
parsuj(l, m);
for (auto& i : m)  std::cout << i.first << " → " << i.second << "\n";
```

Program powinien wypisać:

```
aa → 13
b → Ala=kot
f → xyz
```

6.5 Więcej c-plus-plusa ...

6.5.1 referencje

Referencje są zasadniczo wskaźnikami, których używamy jak zwykłych zmiennych (bez stosowania operatora $*$ w celu operowania na wartości wskaźnika). W odróżnieniu od wskaźników nie możemy bezpośrednio operować na adresie referencji (np. spowodować aby wskazywała na inną zmienną). Kontynuując przykład z modyfikacją zmiennej przekazanej jako argument funkcji, z użyciem referencji kod ten może wyglądać następująco:

```
void ff(int& a) { // zwróć uwagę na & oznaczający że będzie to referencja
    a = 15;
}
int main() {
    x = 10;
    ff(x);
    printf("%d\n", x); // wypisze 15
}
```

6.5.2 iteratory

W powyższych przykładach użycia list i map w C++ warto zwrócić uwagę na użycie iteratorów pozwalających na pobieranie kolejnych wartości z tych kontenerów:

```
void wypiszListe(std::list<int> l) {
    for (std::list<int>::iterator i = l.begin(); i != l.end(); ++i) {
        std::cout << *i << "\n";
    }
}
```

Iterator zwracają niektóre z metod obiektów reprezentujących te kontenery, np. `.begin()` zwraca iterator na pierwszy element. Zwiększanie iteratora odbywa się z użyciem operatorów `++`. Wyjście poza zakres (zwiększenie iteratora wskazującego na ostatni element kolekcji) nie powoduje rzucenia wyjątku, za to iterator przyjmuje specjalną wartość oznaczającą koniec. Iterator o tej wartości zwracany jest przez metodę `.end()` (lub `.rend()` przy iterowaniu w przeciwną stronę).

6.5.3 typ auto

Współczesny C++ oferuje także specjalny typ **`auto`** zwalniający programistę z konieczności jawnego definiowania typu zmiennej do której przypisywana jest od razu jakaś wartość z określonym typem. Możemy napisać np. **`auto`** `x = 5;`, ale nie możemy napisać **`auto`** `x; x = 5;`. Typ ten jest użyteczny np. do obsługi iteratorów, pozwalając powyższą pętle zapisać bez `std::list<int>::iterator` jako:

```
void wypiszListe(std::list<int> l) {
    for (auto i = l.begin(); i != l.end(); ++i) {
        std::cout << *i << "\n";
    }
}
```

6.5.4 pętla for(each)

C++ udostępnia także inną składnię pętli **`for`** pozwalającą na iterowanie po wszystkich elementach kolekcji takich jak listy, mapy, itp. Będącą odpowiednikiem pętli `foreach` znanej z niektórych języków programowania, czy też pythonowskiej pętli **`for`**:

```
void wypiszListe(std::list<int> l) {
    for (auto i : l) {
        std::cout << i << "\n";
    }
}
```

Zamiast **`auto`** `i` możemy napisać **`auto`**`&` `i` aby otrzymać dostęp przez referencję (wtedy wykonanie przypisania wartości do i, np `i = 0`, spowoduje modyfikację elementu listy).

Warto zauważyć także, że w odróżnieniu od wcześniejszej pętli nie operujemy tutaj na iteratorach, a na wartościach / referencjach do wartości z kontenera.

6.5.5 szablony

C++ pozwala też definiować szablony funkcji oraz klas, dzięki którym kompilator będzie mógł wytworzyć funkcje/klasy dla potrzebnych typów w oparciu o ten szablon (zdefiniowany dla ogólnego typu). Na przykład powyższa funkcja wypisująca listy zdefiniowana jest tylko dla list zawierających liczby całkowite. Jednak takie funkcje dla dowolnych typów obsługiwanych przez cout-owy operator `<<` (np. liczb zmiennoprzecinkowych, napisów, ...) będą wyglądały tak samo. Dzięki mechanizmowi szablonów możemy napisać:

```cpp
template <typename T> void wypiszListe(std::list<T>& l) {
    for (auto i : l) {
        std::cout << i << "\n";
    }
}
```

I następnie używać jej dla różnych typów list:

```cpp
int main() {
    std::list<int> x={1, 3, 7, 2, 3};
    wypiszListe(x);

    std::list<float> z={2.7, 5.0, 3.1, 3.9};
    wypiszListe(z);
}
```

Zadanie 6.5.1

Napisz funkcję `wypiszMape` (szablon funkcji) która wypisuje mapę dowolnych typów. Na przykład dla wywołania:

```cpp
std::map<std::string, float> a = { {"xy", 1.3}, {"qw", 16.3} };
std::map<int, std::string> b = { {1, "a"}, {2, "b"} };
wypiszMape(a);
wypiszMape(b);
```

Program powinien wypisać:

```
qw → 16.3
xy → 1.3
1 → a
2 → b
```

6.5.6 lambda

```cpp
int main() {
    int x = 1, y = 1;

    // ta lamba będzie używać:
    //  wartości x z chwili wywołania (i jej zmiana bedzie widoczna na zewnątrz)
    //  wartości y z chwili utworzenia
    auto moja_lambda = [&x, y](int z) { x += z * y; return 11; };

    moja_lambda(2);
    std::cout << x << std::endl; // 3 bo x = 1 + 2 * 1

    x = 0;
    y = 0;

    int z = moja_lambda(2);
    std::cout << x << " " << z << std::endl; // 2 bo x = 0 + 2 * 1
}
```

C++ pozwala także na definiowanie i używanie lambd. Definicja taka składa się z listy przechwytywanych zmiennych, listy argumentów i ciała funkcji. Lista przechwytywania może określać przechwytywanie przez wartość lub przez referencję. W pierwszym przypadku wartość zmiennej z miejsca utworzenia funkcji zostanie w niej "zamrożona", czyli jej dalsze zmiany nie będą widoczne w wywołaniach lambdy.' W drugim przypadku lambda będzie widzieć zawsze aktualną wartość, a zmiany tej zmiennej wewnątrz lambdy będą widoczne także na zewnątrz. Lista argumentów i ciało funkcji działa jak w zwykłych funkcjach. Lambda może zwracać lub może nie zwracać wartość z użyciem return.

6.5.7 jeszcze więcej ... 🤓

Jest to tylko wzmianka o różnych ciekawych aspektach współczesnego C++. Język ten pozwala na dużo więcej (np. przeciążanie operatorów dla naszych typów danych - możemy np. sumować obiekty naszych klas przy pomocy `+`), również biblioteka standardowa oferuje więcej interesujących typów (np. zbiory `std::set`), a używanie szablonów nie sprowadza się jedynie do przedstawionego prostego przypadku szablonu funkcji.

Tematyką C i C++ możnaby wypełnić cały tej długości kurs, a nie jedynie jeden blok zajęć. W naszym kursie ograniczyliśmy się jedynie do omówienia podstawowych zagadnień i krótkiego przeglądu bardziej zaawansowanych możliwości. Zachęcam do nawet pobieżnego zapoznania się np. z „C++ reference" dostępnym pod adresem: https://en.cppreference.com/w/cpp i korzystania z tej dokumentacji przy tworzeniu kodu C++.

7 Wykład wideo[3]

- *Podstawy C* – http://video.opcode.eu.org/06.01-podstawy_C.mkv
- *Funkcje, struktury i tablice w C* – http://video.opcode.eu.org/06.02-C-funkcje_struktury_tablice.mkv
- *Wskaźniki w C* – http://video.opcode.eu.org/06.03-C_wskazniki.mkv
- *Kilka słów o C++* – http://video.opcode.eu.org/06.04-C++.mkv

8 Literatura dodatkowa 🤓

- *Repozytorium z przykładowymi opisanymi kodami C i C++* (https://bitbucket.org/OpCode-eu-org/c_cpp-examples/)
- *Dokumentacja języków C i C++* (https://en.cppreference.com/)
- *Kurs pisania systemu operacyjnego* (https://pl.wikibooks.org/wiki/Pisanie_OS)

9 Rozwiązania zadań

Treści zadań zamieszczone zostały w odpowiednich rozdziałach skryptu.

Poniżej zamieszczone są przykładowe rozwiązania „głównych" zadań z tego skryptu wraz z komentarzami. Wiemy że zajrzenie do nich już przy pierwszej trudności jest kuszące, mimo to rekomendujemy przynajmniej podjąć uczciwą, co najmniej kilkunastominutową na każde z zadań, próbę rozwiązania tych zadania bez zaglądania do odpowiedzi.

Pamiętaj!: Samodzielne rozwiązanie problemu (wraz z wszystkimi trudnościami po drodze i popełnionymi błędami) jest dużo bardziej kształcące od nawet wielokrotnego przepisania gotowego rozwiązania, jednak nawet jednokrotne przepisanie rozwiązania jest bardziej kształcące od wielokrotnego przekopiowania go.

3. Filmy posiadają napisy wgrane do kontenera multimedialnego jako osobny strumień – napisy mogą być włączone lub wyłączone w odtwarzaczu. W wielu filmach dużo dzieje się "na dole ekranu", dlatego polecamy odtwarzać filmy z napisami umieszczonymi poniżej filmu, np. przy pomocy polecenia: `vlc --video-filter='croppadd{paddbottom=120}' --sub-margin=-10 PLIK.mkv`

Rozwiązanie zadania 2.1.1

```
#include <stdio.h>

int main() {
        printf("Hello World\n");
}
```

Zwróć uwagę na:

- wiązanie odpowiedniego pliku nagłówkowego
- umieszczenie kodu wewnątrz funkcji main

Rozwiązanie zadania 2.2.1

```
#include <stdio.h>

int main() {
        for (int i=0; i<10; ++i)
                printf("Hello World\n");
}
```

Zwróć uwagę na użycie pętli :-)

Rozwiązanie zadania 3.3.1

```
#include <stdio.h>

void wypisz(const char* n, int b, int e) {
        for (int i=b; i<=e; ++i)
                printf("%c", n[i]);
        printf("\n");
}

int main(){
        wypisz("i Hello World i", 3, 6);
}
```

Zwróć uwagę na:

- definicję i użycie funkcji przyjmującej argumenty
- iterowanie po elementach tablicy (napisu) i odwołanie się do i-tego elementu operatorem nawias kwadratowego

Rozwiązanie zadania 3.3.2

```
void wypisz(const char* n, int b, int e) {
        for (int i=b; i<=e && n[i]; ++i)
                printf("%c", n[i]);
        printf("\n");
}
```

Zwróć uwagę na dodatkowy warunek w pętli sprawdzający czy aktualny znak nie ma wartości zero (czyli czy nie oznacza końca napisu) łączony operatorem logicznego and z warunkiem dotyczącym numeru znaku do wypisania.

Rozwiązanie zadania 3.3.3

```
#include <stdio.h>

void wypisz(const char* n, int b, int e) {
        int bajt = 0;
```

```
    for (int znak = 0; znak <= e && n[bajt]; ++znak) {
        do {
            if (znak >= b)
                printf("%c", n[bajt]);
            ++bajt;
        } while ((n[bajt] & 0xc0) == 0x80);
    }
    printf("\n");
}

int main(){
    wypisz("i←ĄęĆło World i", 3, 6);
}
```

Rozwiązanie zadania 5.5.1

```
void wypisz(char* n, int b, int e) {
    char* end = n+e;
    for (n=n+b; n<=end && *n; ++n)
        printf("%c", *n);
    printf("\n");
}
```

Zwróć uwagę na:

- zdefiniowanie zmiennej end, będącej wskaźnikiem na ostatni element podnapisu przed pętla
- operację inkrementacji wskaźnika (++n) przy każdym obiegu pętli
- odwołanie do elementu wskazywanego poprzez *n

Rozwiązanie zadania 6.4.1

```
void parsuj(const std::list<std::string>& l, std::map<std::string,std::string>& m)
↪  {
    for (auto& i : l) {
        auto x = i.find("=");
        m[ i.substr(0,x) ] = i.substr(x+1);
    }
}
```

Rozwiązanie zadania 6.5.1

```
#include <iostream>
#include <map>

template <typename A, typename B> void wypiszMape(const std::map<A, B>& m) {
    for (const auto& i : m) {
        std::cout << i.first << " → " << i.second << "\n";
    }
}

int main() {
    std::map<std::string, float> a = { {"xy", 1.3}, {"qw", 16.3} };
    std::map<int, std::string> b = { {1, "a"}, {2, "b"} };
    wypiszMape(a);
    wypiszMape(b);
}
```

Zwróć uwagę na:

- deklaracje funkcji w postaci szablonu z dwoma parametrami - odpowiadającymi parametrom w szablonie jakim jest std::map

- użycie pętli for typu foreach iterującej po elementach otrzymanej mapy
- użycie stałych referencji zarówno w argumencie funkcji, jak i w iteracji po mapie - zapobiega to niepotrzebnemu kopiowaniu mapy i jej elementów