Kurs C

Radosław Łukasik

Wykład 6

O dyrektywach raz jeszcze

Na pierwszym wykładzie wspominaliśmy, że można tworzyć makra za pomocą #define. Makra można również wykorzystać do definiowania typów (jak **typedef**).

```
#define LLINT long lomg int
```

Rozważmy następujące makro i funkcję inline:

```
#define KWADRAT(x) (x)*(x)

inline int kwadrat(int x){
    return x*x;
}
```

Po pierwsze makro nie zawiera typów, więc można wywołać go na każdym typie (zgodnym z działaniami jakie są w makrze)

```
int x=3;
double y=4.5;
x=KWADRAT(x);
y=KWADRAT(y);
```

Po drugie, należy uważać na operatory inkrementacji lub dekrementacji w wywołaniu makra:

```
printf("%d"KWADRAT(x++));// 12, rozwijane do (x++)*(x++)
printf("%d",x);// 5
```

Możliwe jest również tworzenie wielolinijkowych makr ale musimy znakiem backslash dać znać kompilatorowi, że łamiemy linię (po nim nie może być już żadnego znaku w tej linii!). Należy również pamiętać, że w makrze instrukcje oddziela się średnikiem tak jak zwykle, jedynie po ostatniej instrukcji nie musimy go dawać. W makrach nie można używać dyrektyw procesora, ale można używać zdefiniowanych makr. Mamy również dostępne pewne makra: **__FILE__** - zwraca łańcuch znaków będący pełną ścieżką do pliku źródłowego z którego pochodzi wywołanie, **__DATE__** i **__TIME__** - zwracają łańcuchy znaków zawierające odpowiednio datę i czas kompilacji programu, **__LINE__** - zwraca liczbę całkowitą będącą numerem linii w której znajduje się użycie tego makra. Numer linii może być przydatny podczas testowania programu (pewien rodzaj debugowania), **__func__** - zwraca nazwę funkcji w której wywołano makro.

```
#define DEBUG // wystarczy usunąć tą linie by nic się nie wyświetlało
#define ctime() printf("%s\n%s %s\n", __DATE__,__TIME__);
#ifdef DEBUG
    #define log_int(X)  \
        printf("%s, %s:%d %s=%d\n",__FILE__,_FUNC__,__LINE__,#X,X)
#else
    #define log_int(X)
#endif
int zmienna=5;
log_int(zmienna);
```

We wcześniejszym przykładzie pojawiło się w wyświetlaniu użycie znaku #. Jest to operator preprocesora służący zamianie (wydobycie) nazwy zmiennej na łańcuch. Dzięki temu nie musimy sami wpisywać nazwy tej zmiennej. Mamy dostępny tutaj jeszcze drugi operator - podwojony hasz oznaczający konkatenację, służący do łączenia dwóch łańcuchów znaków.

Przykład

```c
#include <stdio.h>
#include <limits.h>
#include <float.h>
#define STR(x) #x
#define zrob__(x) STR(__## x ##__)
#define MAX_int INT_MAX
#define MIN_int INT_MIN
#define MAX_double DBL_MAX
#define MIN_double DBL_MIN
#define DISPLIM(T,L) printf(STR(L)" "STR(T)": %g\n",(double)L##_##T)
int main(){
    DISPLIM(int,MIN);
    DISPLIM(double,MAX);
    printf("%s",zrob__(Kurs C));
    return 0;
}
```

W makrach możemy tworzyć również zmienne lokalne ale należy całą definicję makra umieścić w nawiasach klamrowych. Użycie zmiennej lokalnej bez nawiasów klamrowych nie pozwala na użycie tego makra po raz drugi w obrębie tego samego bloku co pierwsze wywołanie tego makra, a także powoduje zajęcie miejsca na stosie cały czas w obrębie tego bloku.

```
#define POTEGA(a,n) \
{ \
  int temp = a; \
  for(int i=0;i<n;i++) \
  a*=temp; \
}
```

Powyższe makro nie skompiluje się gdy 'a' nie będzie zmienną

```
int x=2;
POTEGA(3,4);// błąd kompilacji
POTEGA(x+2,4);// błąd kompilacji
POTEGA(++x,4);// błąd kompilacji
POTEGA(x,4);// bez błędu
```

Pamiętajmy o nawiasach dla makr, tam gdzie to konieczne

```
#define KWADRAT1(x) x*x // niepoprawnie
#define KWADRAT2(x) (x)*(x) // poprawnie
int x=3;
printf("%d\n",KWADRAT1(x+1));// 3+1*3+1=7
```

Dyrektyw możemy używać do pisania programu na różne systemy operacyjne:

```
#ifdef __linux__
// używamy funkcji dostępnych tylko w linux
#elif _WIN32
// używamy funkcji dostępnych tylko w windows
#endif
```

Przykład

```
#ifdef __linux__
// używamy funkcji dostępnych tylko w linux
    #include <time.h>
    void ms_sleep(unsigned czas){
        struct timespec req;
        req.tv_sec = (czas / 1000);
        req.tv_nsec = (czas % 1000 * 1000000);
        nanosleep(&req, NULL);
    }
#elif _WIN32
// używamy funkcji dostępnych tylko w windows
    #include <windows.h>
    void ms_sleep(unsigned czas){
        Sleep(czas);
    }
#endif // dla innych systemów brak funkcji
```

Przykład - uzyskanie liczby wątków procesora

```
#ifdef _WIN32
    #include <windows.h>
    int NumThreads(int *dostepne,int *maksymalnie){
        SYSTEM_INFO info; GetSystemInfo(&info);
        *maksymalnie=info.dwNumberOfProcessors;
        *dostepne=info.dwNumberOfProcessors;
        return 0;
    }
#else
    #include <unistd.h>
    int NumThreads(int *dostepne,int *maksymalnie){
        *maksymalnie=1; *dostepne=1;
    #ifdef _SC_NPROCESSORS_ONLN
        *dostepne=sysconf(_SC_NPROCESSORS_ONLN);
        *maksymalnie=sysconf(_SC_NPROCESSORS_CONF);
        return 0;
    #else
        return 1;
    #endif
    }
#endif
```

Optymalizacje przy kompilacji

Współczesne kompilatory pozwalają na wybranie pewnych opcji podczas kompilowania programu. Mogą mieć one wpływ na rozmiar pliku wykonywalnego jak i na szybkość jego działania. Dla kompilatora gcc mamy dostępnych sporo opcji. Głównie interesują nas **-O, -O1, -O2, -O3** oznaczające optymalizacje pod względem szybkości (im większa cyfra tym lepsza optymalizacja, **-O2** może zmieniać kolejność instrukcji, **-O3** dodatkowo może zwiększać rozmiar pliku poprzez techniki podobne do funkcji **inline**) oraz **-Os** oznaczające optymalizację pod względem rozmiaru (okazuje się, że rozmiar również wpływa na szybkość). Uwaga! Optymalizacje **-O2** i **-O3** mogą być w pewnych przypadkach wolniejsze niż **-O**.

Może się zdarzyć, że optymalizacja wytnie nam kawałek kodu, gdyż kompilator uzna, że po jego wykonaniu nie będziemy korzystać w żaden sposób z danych otrzymanych (np. mamy wyzerowanie tablicy dynamicznej a zaraz potem jej zwolnienie). Może się również zdarzyć w bardzo specyficznych sytuacjach, że kod stanie się niepoprawny (dotyczy to wstawek asemblerowych, korzystania w sposób niebezpośredni ze stosu, wielowątkowości). Wprawdzie istnieje modyfikator zmiennych **volatile**, który ma za zadanie chronić zmienne przed błędną optymalizacją ale nie zawsze się to udaje.

Przykład gdy lepiej nie optymalizować

```c
#include <stdio.h>
// to zadziała bez optymalizacji
void f1(int a){
    int x;// tworzymy na stosie zmnienna
    x=a++;// i przypisujemy jej wartość a
}
void f2(){
    int y;// zmienna y jest w tym samym miejscu w pamięci co x !!!
    printf("y=%d\n",y);// wyświetli się 3 !!!
}
int main(){
    f1(3);// na stosie odkładamy adres powrotu funkcji oraz liczbę 3
    f2();// na stosie odkładamy adres powrotu
    int z;// kompilator nie jest dla nas łaskawy, zmienna z
    printf("z=%d\n",z);// nie jest tworzona w tym miejscu kodu !
    return 0;
}
```

Porównanie optymalizacji

Optymalizacje - brak	
kopiowanie for up	4.320337
kopiowanie memcpy	0.067074
wypelnianie kwadratami	4.640432
funkcja o2	4.626720
sort. kubelkowe	5.342274

Optymalizacje - Os	
kopiowanie for up	0.825129
kopiowanie memcpy	0.079079
wypelnianie kwadratami	1.134947
funkcja o2	0.678669
sort. kubelkowe	3.274801

Optymalizacje - O	
kopiowanie for up	0.733816
kopiowanie memcpy	0.079361
wypelnianie kwadratami	1.659832
funkcja o2	0.695515
sort. kubelkowe	2.305765

Optymalizacje - O1	
kopiowanie for up	0.733448
kopiowanie memcpy	0.079304
wypelnianie kwadratami	1.619387
funkcja o2	0.678526
sort. kubelkowe	2.225285

Optymalizacje - O2	
kopiowanie for up	1.301853
kopiowanie memcpy	0.079168
wypelnianie kwadratami	1.079594
funkcja o2	1.079596
sort. kubelkowe	2.151979

Optymalizacje - O3	
kopiowanie for up	0.096827
kopiowanie memcpy	0.079172
wypelnianie kwadratami	0.208675
funkcja o2	0.209515
sort. kubelkowe	2.357161

```
int o2(int a){// w O3 inline
    int x,y;
    x=a;
    x=x+1;
    y=a;
    y=y+1;
    return (x+y);
}
```

```
int kwadrat(int a){// w O3 inline
    return a*a;
}
```

Sprawdzanie poprawności kodu

Gdy chcemy testować działanie programu, to może nam się przydać kilka rzeczy. Pierwsza z nich polega na podmianie dowolnej funkcji na napisaną przez nas (np. symulacja wejścia, danych z sieci, itp.):

```c
#include <stdio.h>
#include <time.h>
#define PODMIANA // włączamy podmianę

#ifdef __PODMIANA__
time_t my_time(time_t *czas){
    if(czas) *czas=5;
    return 5;
}
#define time my_time
#endif

int main(){
    time_t czas=time(0);
    printf("Czas %ld\n",czas);
    return 0;
}
```

Inną metodą testowania jest użycie asercji (biblioteka **assert.h**) i dostępna w nich funkcja **assert(wyrażenie)**, która testuje podane wyrażenie i jeśli ono zachodzi, to program wykonywany jest dalej. W przeciwnym wypadku program kończy swe działanie wypisując w standardowe wyjście błędów testowane wyrażenie wraz z linią i nazwą pliku źródłowego w którym występowała asercja.
Jeżeli chcemy wyłączyć asercję w całym kodzie, po zakończeniu testowania w wersji wydawniczej, to można to zrobić dodając #define NDEBUG przed dodaniem biblioteki **assert.h**.

Przykład

```c
#include <stdio.h>
//#define NDEBUG // wyłącznik asercji
#include <assert.h>
int main(){
    int x=5;
    assert(x==4);
    printf("Udalo sie\n");
    return 0;
}
```

Asercja może być przydatna w miejscach w których mamy obawy przed niedozwolonymi działaniami (np. dzielenie przez 0, błędy we wskaźnikach). Pamiętajmy by nie stosować asercji zamiast sprawdzania czy wskaźnik nie jest niezerowy!

W tej samej bibliotece mamy również funkcje do testowania kodu podczas kompilacji, a nie podczas działania jak było przed chwilą. Jest to **static_assert(wyrażenie,komunikat)**, która testuje podane wyrażenie i jeśli ono zachodzi, to program wykonywany jest dalej. W przeciwnym wypadku program nie kompiluje się wypisując podany komunikat.

Przykład

```
#include <stdio.h>
#include <assert.h>
int main(){
    static_assert(sizeof(void*) == 8,"Komiluj tylko na 64 bitowej
    platformie");
    printf("Dziala\n");
    return 0;
}
```

Debugger

Debugger jest narzędziem pozwalającym śledzić działanie programu (np. wartości zmiennych, przechodzenie programu przez pewne miejsca w kodzie) podczas jego wykonywania.
Aby używać w Code::Blocks debuggera należy program, który chcemy testować utworzyć jako projekt (**File->New->Project, Console Application**) z utworzonymi konfiguracjami zarówno "**Debug**" jak i "**Release**". Wprawdzie każdy projekt pozwala na ustawienie opcji kompilatora, ale niestety trzeba przestawić główne opcje by móc testować program. W tym celu należy wejść w **Settings->Compiler** i na zakładce **Compiler settings** sprawdzić czy są odznaczone wszystkie opcje z sekcji **Optimization**, a także zaznaczone wszystkie opcje z sekcji **Debugging**. Należy też uważać na nazwę projektu i ścieżkę do niego, bo znaki spoza alfabetu angielskiego mogą również powodować problemy.

Na pasku narzędzi możemy znaleźć (jeśli nie ma ich tam, to da się włączyć) przyciski związane z debuggerem:

- Debug/Continue - Uruchamia lub wznawia działanie debuggera;
- Run to cursor - przechodzi do odpluskwiania w miejscu gdzie aktualnie znajdował się kursor w kodzie źródłowym;
- Next Line - przechodzi do następnej linii kodu (nie wchodzi do wnętrza funkcji);
- Step into - przechodzi do kolejnej linii kodu, gdy w kodzie w aktualnym miejscu mamy wywołanie funkcji, to pozwala nam na wejście do niej;
- Step out - powoduje wyjście z debuggowania kodu funkcji do której weszliśmy za pomocą Step into lub Step into instruction;
- Next instruction - następna instrukcja procesora (nie jest to instrukcja z C++ lecz z kodu przetłumaczonego na język assembler);
- Step into instruction - przechodzi po kolejnych instrukcjach procesora, ale wchodzi również do funkcji, gdy są one wywoływane;
- Break debugger - wstrzymuje działanie debuggera.
- Stop debugger - zatrzymuje działanie debuggera.

Oprócz tego na pasku zadań mamy przycisk **Debugging windows**, po jego kliknięciu pojawia się menu, z którego możemy wybrać:

- Breakpoints - wyświetla listę wszystkich linii kodu gdzie użytkownik chciał przerwać program;
- CPU Registers - pokazuje aktualne wartości rejestrów procesora;
- Call stack - pokazuje nazwy aktualnie wywołanych funkcji w programie (zawsze jest tam funkcja **main()**);
- Disassembly - pokazuje asemblerowy kod odpowiadający kodowi programu wraz z zaznaczonym aktualnym miejscem wykonywania programu i adresem w pamięci;
- Memory dump - pozwala podejrzeć aktualną zawartość pamięci komputera (należy podać adres bezpośrednio lub adres z nazwą zmiennej);
- Running threads - pokazuje listę działających wątków związanych z debugowanym programem;
- Watches - wyświetla listę wszystkich obserwowanych zmiennych i ich aktualnych wartości.

Aby utworzyć punkt przerwania należy kliknąć po prawej stronie numeru linii kodu, w którym chcemy umieścić taki punkt. Pojawi się wówczas czerwona kropka. Ponowne kliknięcie usuwa punkt przerwania.
Zmienne, które chcemy śledzić dodajemy dopiero po uruchomieniu debuggera. Wystarczy kliknąć prawym przyciskiem myszy na nazwę zmiennej i wybrać "watch" z nazwą zmiennej. Zmienną również można usunąć z listy obserwowanych (w okienku **Watches**).
Pamiętajmy o tym, że makr nie można debugować! Pluskwa.cbp

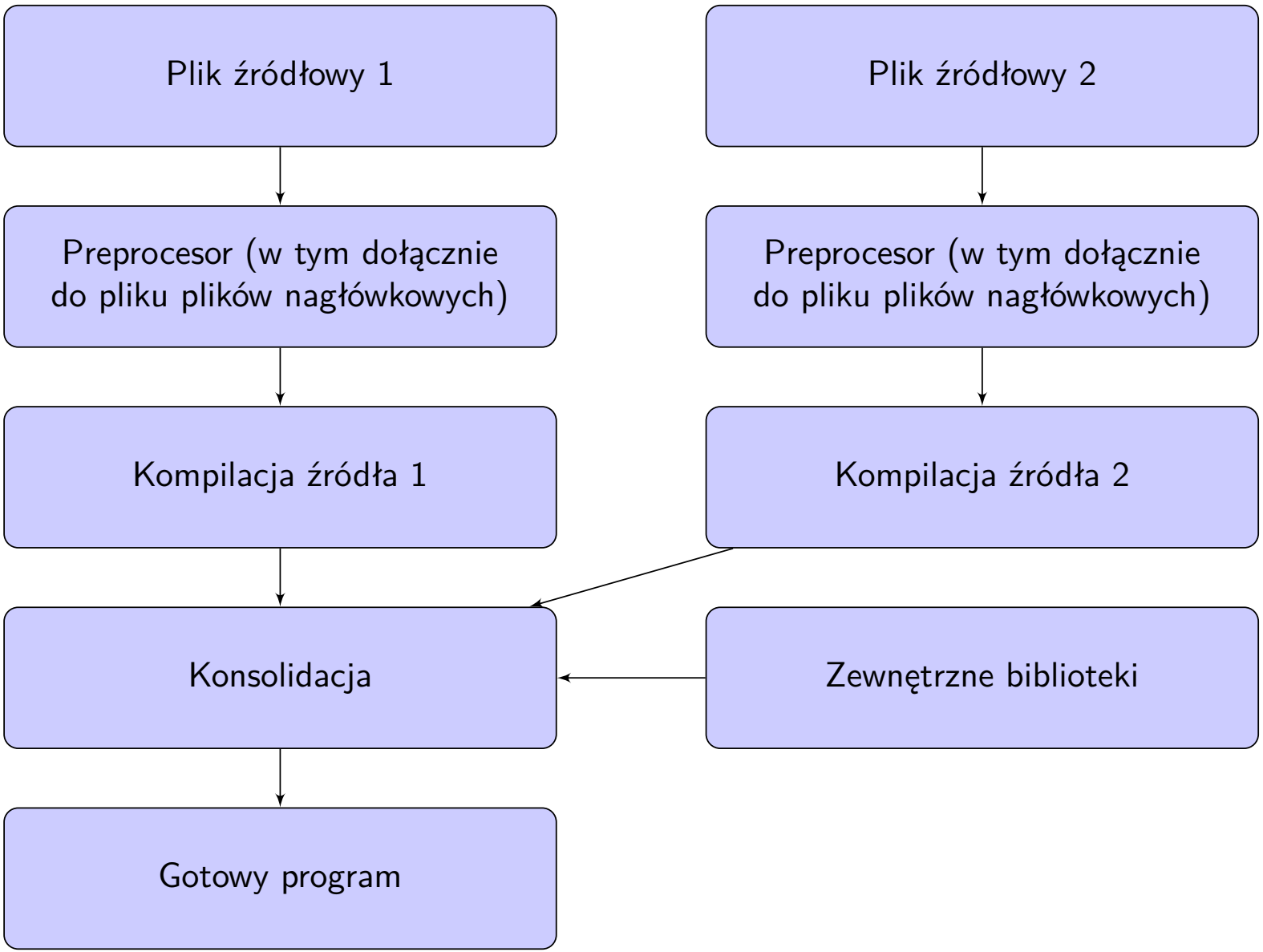
Pliki źródłowe i nagłówkowe

W przypadku dużych projektów dobrze jest podzielić go na części, by łatwo było się połapać w kodzie. Za pomocą #include + cudzysłów możemy dołączać te pliki, przy czym wyróżnia się ich dwa rodzaje: pliki źródłowe (rozszerzenie ".c") - zawierają definicję funkcji i deklarację zmiennych oraz pliki nagłówkowe (rozszerzenie ".h") - zawierają definicję stałych, typów, struktur, funkcji inline i deklarację funkcji (domyślnie dostępnych w pliku źródłowym o tej samej nazwie, plik źródłowy zawiera odwołanie do pliku nagłówkowego a nie odwrotnie!, główny plik programu może zawierać odwołania tylko do plików nagłówkowych jeżeli funkcje te dostępne są w innym pliku pośrednim - z rozszerzeniem ".o" - lub zewnętrznej bibliotece dołączanych podczas linkowania). Ponieważ pliki nagłówkowe mogą być dołączane w innych plikach źródłowych, czy też nagłówkowych, to warto zabezpieczyć się przed kolejną kompilacją tego samego pliku (kończącą się przeważnie błędem). W tym celu używa się wartowników:

```
#ifndef NAZWAZRODLA_H
#define NAZWAZRODLA_H
// zawartość pliku nagłówkowego
#endif
```

Dla plików źródłowych przeważnie takie zabezpieczenie nie jest potrzebne bo odwołanie do nich jest poprzez deklaracje w plikach nagłówkowych o tej samej nazwie.

Schemat tworzenia programu



Warto tutaj wspomnieć o opcji kompilowania -**static** (statyczne linkowanie), która to wymusza by funkcje nie były wywoływane z zewnętrznych bibliotek, tylko włączone w kod programu. Zwiększa to rozmiar pliku wykonywalnego, ale uniezaleznia nas od istnienia tych bibliotek.
Normalnie tworzone zmienne z innych plików źródłowych nie są dostępne poza obrębem funkcji z tego pliku. Jeżeli chcielibyśmy to zmienić, to można dodać modyfikator **extern** przed typem zmiennej. Funkcje zdeklarowane w pliku nagłówkowym domyślnie są widoczne na zewnątrz, więc nie jest potrzebne dla nich modyfikator **extern**. Jeżeli chcemy by funkcja była dostępna tylko dla użytku danego modułu, to nie należy umieszczać jej deklaracji w pliku nagłówkowym a tylko w źródłowym (generuje ostrzeżenie). Natomiast poprzedzenie takiej funkcji słowem static całkiem ogranicza jej dostęp na zewnątrz.

Przykład

```
// plik funkcje.h
#ifndef FUNKCJE_H
#define FUNKCJE_H
int mojafunkcja(int x);
extern int licznik;
#endif
```

Przykład

```c
// plik funkcje.c
int licznik=0;
int licznik2=0;

static int wewnetrzna(int x){
    x+=licznik++;
    licznik2++;
    return x;
}

int mojafunkcja(int x){
    licznik++;
    return wewnetrzna(x)+1;
}
```

```c
// plik main.c
#include <stdio.h>
#include "funkcje.h"
int main(){
    int y=mojafunkcja(4);
    printf("%d, licznik= %d\n",y,licznik);
    return 0;
}
```

Fizz Buzz

- Program, który w zakresie od 1 do n wypisuje tę liczbę lub zamiast niej:
  - "Fizz", gdy jest ona wielokrotnością 3;
  - "Buzz", gdy jest wierokrotnośćią 5;
  - "FizzBuzz", gdu jest wielokrotnością 15.

Fizz Buzz

- Program, który w zakresie od 1 do n wypisuje tę liczbę lub zamiast niej:
  - "Fizz", gdy jest ona wielokrotnością 3;
  - "Buzz", gdy jest wierokrotnośćią 5;
  - "FizzBuzz", gdu jest wielokrotnością 15.

```
bool isPrinted;
for(int i=1; i<=n; i++){
    isPrinted = false;
    if(i%3 == 0){
        isPrinted = true;
        printf("Fizz");
    }
    if(i%5 == 0){
        isPrinted = true;
        printf("Buzz");
    }
    if(!isPrinted)
        printf("%d",i);
    printf("\n")
}
```

```
for(int i=1; i<=n; i++){
    if(i%15 == 0){
        printf("FizzBuzz");
    }else{
        if(i%3 == 0){
            printf("Fizz");
        }else{
            if(i%5 == 0)
                printf("Buzz");
            else
                printf("%d",i);
        }
    }
    printf("\n")
}
```

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

```
int tablica[]={1,2,3,4,5,6};
printf("L. elementow: %zu",sizeof(tablica)/sizeof(tablica[0]));
```

Nie da się tego zrobić dla tablic dynamicznych, bo ich rozmiar nie jest znany w momencie kompilacji!

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

```
int tablica[]={1,2,3,4,5,6};
printf("L. elementow: %zu",sizeof(tablica)/sizeof(tablica[0]));
```

  Nie da się tego zrobić dla tablic dynamicznych, bo ich rozmiar nie jest znany w momencie kompilacji!

- Co jest nie tak z poniższym kodem?

```
int main{
    int *i;
    *i = 10;
}
```

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

```
int tablica[]={1,2,3,4,5,6};
printf("L. elementow: %zu",sizeof(tablica)/sizeof(tablica[0]));
```

  Nie da się tego zrobić dla tablic dynamicznych, bo ich rozmiar nie jest znany w momencie kompilacji!

- Co jest nie tak z poniższym kodem?

```
int main{
    int *i;
    *i = 10;
}
```

  Adres we wskaźniku jest losowy, co może powodować błąd działania programu (naruszenie ochrony pamięci).

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

```
int tablica[]={1,2,3,4,5,6};
printf("L. elementow: %zu",sizeof(tablica)/sizeof(tablica[0]));
```

  Nie da się tego zrobić dla tablic dynamicznych, bo ich rozmiar nie jest znany w momencie kompilacji!

- Co jest nie tak z poniższym kodem?

```
int main{
    int *i;
    *i = 10;
}
```

  Adres we wskaźniku jest losowy, co może powodować błąd działania programu (naruszenie ochrony pamięci).

- Co jest nie tak z poniższym kodem?

```
int main(){
    char *array="hello, world";
    array[0]='A';
}
```

- Jak uzyskać ilość elementów statycznej tablicy? Czy da się coś takiego zrobić z tablicą dynamiczną?

```
int tablica[]={1,2,3,4,5,6};
printf("L. elementow: %zu",sizeof(tablica)/sizeof(tablica[0]));
```

  Nie da się tego zrobić dla tablic dynamicznych, bo ich rozmiar nie jest znany w momencie kompilacji!

- Co jest nie tak z poniższym kodem?

```
int main{
    int *i;
    *i = 10;
}
```

  Adres we wskaźniku jest losowy, co może powodować błąd działania programu (naruszenie ochrony pamięci).

- Co jest nie tak z poniższym kodem?

```
int main(){
    char *array="hello, world";
    array[0]='A';
}
```

  Adres we wskaźniku jest w pamięci tylko do odczytu, co powoduje błąd działania programu (naruszenie ochrony pamięci).

- Jaka wartość się wyświetli?

```
int main(){
    int i = 9;
    int j = 0;
    switch(i){
      case 9:
        j += 200;
      case 10:
        if(j==0){
            j+=300;
           break;
      case 11: ;
        } else {
            j+=100;
        }
       break;
      default: ;
    }
    printf("%d",j);
}
```

- Jaka wartość się wyświetli?

```
int main(){
    int i = 9;
    int j = 0;
    switch(i){
      case 9:
        j += 200;
      case 10:
        if(j==0){
            j+=300;
           break;
      case 11: ;
        } else {
            j+=100;
        }
        break;
      default: ;
    }
    printf("%d",j);
}
```

300, może dziwić, że w ogóle się to kompiluje, ale w C można tak mieszać **if**..**else** z przypadkami w **switch**.

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

0, pod if-em pierwszy warunek nie jest niezerowy, więc do drugiego już nie przechodzimy.

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

0, pod if-em pierwszy warunek nie jest niezerowy, więc do drugiego już nie przechodzimy.

- Czym różnią się te dwa wyrażenia? Dla jakich a i b otrzymamy różne wyniki?

```
float a, b;
// jakiś kod przypisujący a i b pewne wartości
float wyrazenie_1 = 1.0 + a + b;
float wyrazenie_2 = 1.0f + a + b;
```

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

0, pod if-em pierwszy warunek nie jest niezerowy, więc do drugiego już nie przechodzimy.

- Czym różnią się te dwa wyrażenia? Dla jakich a i b otrzymamy różne wyniki?

```
float a, b;
// jakiś kod przypisujący a i b pewne wartości
float wyrazenie_1 = 1.0 + a + b;
float wyrazenie_2 = 1.0f + a + b;
```

W pierwszym wyrażeniu 1.0 jest traktowane jako **double** i dalsze dodawanie jest właśnie tego typu. Na koniec dopiero mamy rzutowanie do **float**. Wystarczy wziąć $a = b = 2^{-24}$ (bo mantysa w typie **float** ma 23 bity).

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

0, pod if-em pierwszy warunek nie jest niezerowy, więc do drugiego już nie przechodzimy.

- Czym różnią się te dwa wyrażenia? Dla jakich a i b otrzymamy różne wyniki?

```
float a, b;
// jakiś kod przypisujący a i b pewne wartości
float wyrazenie_1 = 1.0 + a + b;
float wyrazenie_2 = 1.0f + a + b;
```

W pierwszym wyrażeniu 1.0 jest traktowane jako **double** i dalsze dodawanie jest właśnie tego typu. Na koniec dopiero mamy rzutowanie do **float**. Wystarczy wziąć $a = b = 2^{-24}$ (bo mantysa w typie **float** ma 23 bity).

- Jak odczytać łańcuch znaków nie powodując przepełnienia bufora do którego wczytujemy?

- Jaka wartość się wyświetli?

```
int main(){
    int i = 0;
    int j = 0;
    if ( j && ++i ) ;
    printf("%d",i);
}
```

0, pod if-em pierwszy warunek nie jest niezerowy, więc do drugiego już nie przechodzimy.

- Czym różnią się te dwa wyrażenia? Dla jakich a i b otrzymamy różne wyniki?

```
float a, b;
// jakiś kod przypisujący a i b pewne wartości
float wyrazenie_1 = 1.0 + a + b;
float wyrazenie_2 = 1.0f + a + b;
```

W pierwszym wyrażeniu 1.0 jest traktowane jako **double** i dalsze dodawanie jest właśnie tego typu. Na koniec dopiero mamy rzutowanie do **float**. Wystarczy wziąć $a = b = 2^{-24}$ (bo mantysa w typie **float** ma 23 bity).

- Jak odczytać łańcuch znaków nie powodując przepełnienia bufora do którego wczytujemy? Realokacja pamieci bufora.

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

Było na wykładzie!

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

Było na wykładzie!

- Co się wyświetli?

```
float x = 4.6;
if(x == 4.6)
    printf("1");
else
    printf("2");
```

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

Było na wykładzie!

- Co się wyświetli?

```
float x = 4.6;
if(x == 4.6)
    printf("1");
else
    printf("2");
```

2, pod if mamy konwersję x na double (bo tak rozumiemy 4.6), a 4.6f != 4.6 (błąd przybliżeń).

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

Było na wykładzie!

- Co się wyświetli?

```
float x = 4.6;
if(x == 4.6)
    printf("1");
else
    printf("2");
```

2, pod if mamy konwersję x na double (bo tak rozumiemy 4.6), a 4.6f != 4.6 (błąd przybliżeń).

- Gdzie mogą być przechowywane dane?

- Jaka jest różnica między tymi trzema wskaźnikami?

```
const char * p1;
char * const p2;
char const * p3;
```

Było na wykładzie!

- Co się wyświetli?

```
float x = 4.6;
if(x == 4.6)
    printf("1");
else
    printf("2");
```

2, pod if mamy konwersję x na double (bo tak rozumiemy 4.6), a 4.6f != 4.6 (błąd przybliżeń).

- Gdzie mogą być przechowywane dane?
  Sterta, stos, segment danych (zainicjowane zmienne globalne i statyczne zmienne lokalne), segment BSS (niezainicjowane zmienne globalne i statyczne zmienne lokalne) - zmienne w nim są inicjowane zerami, segment kodu (lub inaczej tekstowy, tylko do odczytu!).