Języki i metody programowania I

dr inż. Piotr Szwed
Katedra Informatyki Stosowanej
C2, pok. 403

e-mail: pszwed@agh.edu.pl

http://home.agh.edu.pl/~pszwed/

Aktualizacja: 2013-01-18

9. Struktura programów

Deklaracje

W języku C/C++ występują dwa typy deklaracji:

- **Definicje** – pociągają za sobą przydział pamięci dla zmiennej
- **Referencje** – informują kompilator o pojawieniu się identyfikatora zmiennej (lub funkcji) określonego typu. Sama definicja występuje później lub w innym module kodu źródłowego.
- Przykład definicji:

```
int x;
```

Wprowadza identyfikator o nazwie x, przydziela pamięć dla zmiennej.

- Przykład deklaracji będącej referencją:

```
extern int x;
```

Wprowadza identyfikator o nazwie `x`, pamięć dla zmiennej nie jest przydzielana. Nastąpi to później lub zmienna pojawi się w innym module.

Czas życia (1)

- Czas życia to okres w trakcie wykonania programu, kiedy zmienna lub funkcja istnieją, czyli jest dla nich przydzielone miejsce w pamięci.
- Ze względu na czas życia identyfikatory dzielimy na:
  - statyczne (w tym globalne),
  - automatyczne (lokalne).
- W praktyce, wszystkie funkcje istnieją podczas wykonania programu. Kod funkcji jest załadowany do pamięci w momencie uruchamiania programu i istnieje przez cały czas jego działania (wyjątkiem są dynamicznie linkowane biblioteki DLL).

Czas życia (2)

Zmienne o statycznym przydziale pamięci

- Pamięć dla zmiennych statycznych jest zarezerwowana na stałe w trakcie działania programu.
- Zmienne te są inicjowane wartościami początkowymi tylko raz, przed rozpoczęciem wykonania programu. Jeżeli w momencie definicji zmienna statyczna nie zostanie jawnie zainicjowana, wówczas standardowo nadana jej zostanie wartość 0.
- Aby zadeklarować zmienną o statycznym przydziale pamięci:
  - Deklarujemy ją poza blokami funkcji, na tym samym poziomie co definicje funkcji; zmienne te nazywane są **globalnymi**.
  - Używamy słowa kluczowego `static`

Czas życia (3)

Zmienne automatyczne

- Zmiennymi automatycznymi są wszystkie zmienne zadeklarowane wewnątrz funkcji lub wewnątrz instrukcji blokowej, o ile nie są poprzedzone modyfikatorem `static`.
- Wszystkie parametry funkcji są również zmiennymi automatycznymi.
- Czas życia zmiennych automatycznych jest powiązany z wykonaniem instrukcji blokowej {...}.
  Pamięć dla zmiennych automatycznych jest przydzielana w momencie wejścia do instrukcji blokowej i zwalniana w momencie wykonania ostatniej instrukcji bloku.
- Zmienne automatyczne powinny być inicjowane przy każdym wejściu do bloku. Jeżeli nie nadamy zmiennej wartości początkowej, wówczas będzie ona miała wartość nieokreśloną.

Czas życia (4)

Zmienne automatyczne …

- Pamięć dla zmiennych automatycznych przydzielana jest na stosie.
- Wyjątkiem są zmienne rejestrowe. Zmienne rejestrowe deklarujemy z użyciem słowa kluczowego `register`. Należy je traktować jako **wskazówkę** dla kompilatora, aby odwzorowywał zmienną w jeden z wolnych rejestrów procesora, zamiast w komórkę pamięci. Brak jest jednak gwarancji, że żądanie to zostanie zrealizowane.

Przykład 1

```
int a=1; // zmienna globalna

void f1()
{
 int x = 2;          // zmienna lokalna (automatyczna)
 static int y=0;     // zmienna statyczna
 x--; y++;a*=2;
 printf("x=%d y=%d a=%d; ", x, y, a) ;
}

int  main()
{
 f1() ;
 f1() ;
 f1() ;
 return 0;
}
```

```
x=1 y=1 a=2; x=1 y=2 a=4; x=1 y=3 a=8;
```

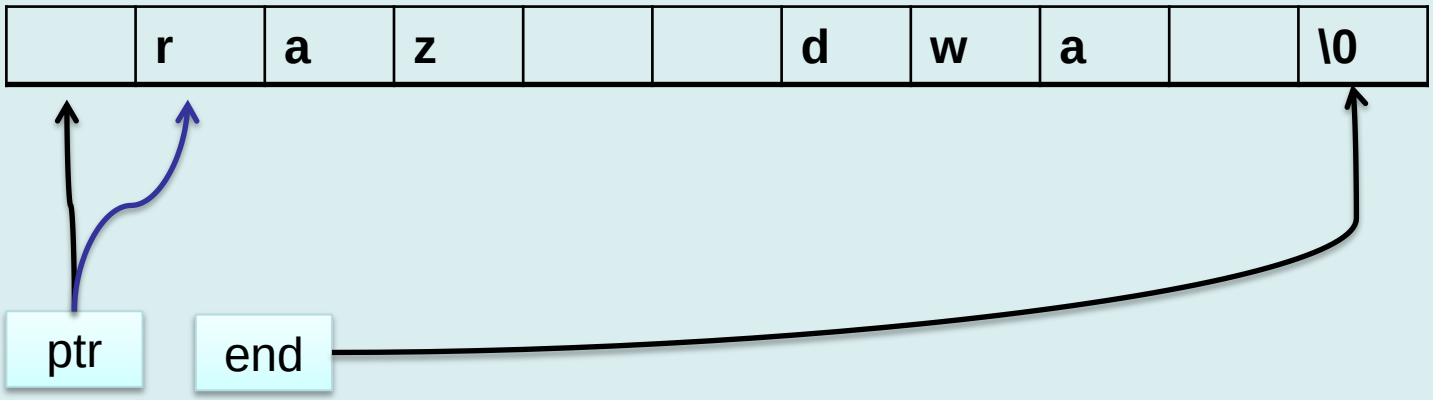
Przykład 2

```
char*tokenize(char*txt, const char*sep)
{
  static char*ptr=0;
  static char*end=0;
  char*ret=0;

  if(txt!=0){
    ptr = txt;
    end=txt+strlen(txt);
    while(ptr<end && *ptr && strchr(sep,*ptr))ptr++;
  }
  if(ptr>=end || *ptr==0 )return 0;
  ret = ptr;
  while(ptr<end && *ptr && !strchr(sep,*ptr))ptr++;
  while(ptr<end && *ptr && strchr(sep,*ptr)){*ptr=0;ptr++;}
  return ret;
}
```

Przykład 2 (cd)

Faza początkowa – ustawianie wskaźników `ptr` i `end`, pomijanie początkowych separatorów

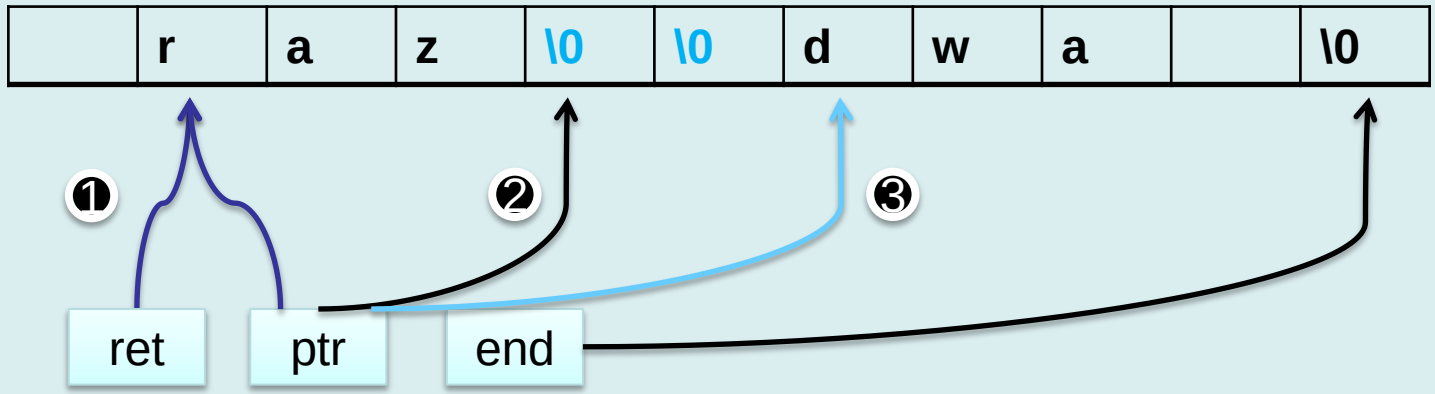



```
static char*ptr=0;
static char*end=0;
char*ret=0;

if(txt!=0){
    ptr = txt;
    end=txt+strlen(txt);
    while(ptr<end && *ptr && strchr(sep,*ptr))ptr++;
}
```

Przykład 2 (cd)

Faza druga – wydzielanie symboli




```
    if(ptr>=end || *ptr==0 )return 0;
(1) ret = ptr;
(2) while(ptr<end && *ptr && !strchr(sep,*ptr))ptr++;
(3) while(ptr<end && *ptr && strchr(sep,*ptr))
    {*ptr=0;ptr++;}
    return ret;
```

Przykład 2 (cd)

Wywołanie

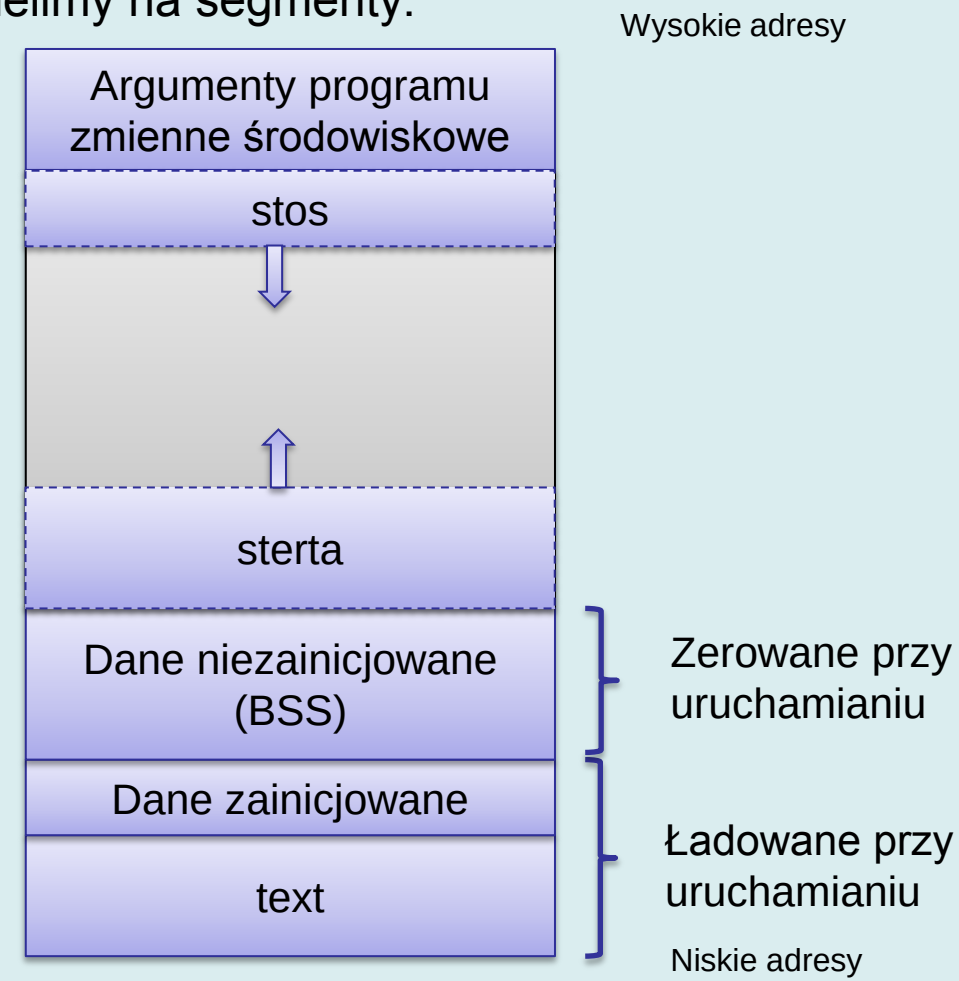
```
int main(){
    char txt[256] = " Ala ma , . kota....i psa ";

    char*token;
    for(      token=tokenize(txt," .,");
              token;
              token=tokenize(0," .,"))
    {
       printf("%s\n",token);
    }
    return 0;
}
```

```
Ala
ma
kota
i
psa
```

Przydział pamięci dla programu (1)

- Podczas uruchamiania zapisany na dysku program staje się procesem.
- Kod programu ładowany do pamięci RAM. Dodatkowo przydzielana jest pamięć dla stosu i sterty.
- Pamięć programu dzielimy na segmenty.



Przydział pamięci dla programu (2)

- Segment text (kod)
  - Zawiera skompilowany do postaci wykonywalnych rozkazów maszynowych kod funkcji.
  - Nie jest możliwa modyfikacja zawartości segmentu kodu w trakcie działania programu (read-only).
  - Dla bezpieczeństwa umieszczany przy niskich adresach pamięci
  - W przypadku wielodostępu, segment kodu może być dzielony przez różne instancje procesów
- Segment danych (zainicjowanych)
  - Zawiera zmienne o statycznym czasie życia: **globalne** i zadeklarowane lokalnie z użyciem modyfikatora `static`.
  - Obejmuje dane tylko do odczytu (ang. read-only - RO) oraz dane do odczytu i zapisu (ang. read-write - RW)

```
char*ptr="Hello world";    // ptr w RW; "Hello world" w RO
char tab[]="Ala ma kota"; // tab w RW zaincjowane tekstem
                           // "Ala ma kota"
int count=0;               // count w RW
```

Przydział pamięci dla programu (3)

- Segment niezainicjowanych danych (BSS)
  - Dane w tym segmencie są zerowane przed rozpoczęciem wykonania programu.
  - Nazwa pochodzi od rozkazu asemblera z 1955 roku *block started by symbol*, ale niektórzy tłumaczą, jako *better save space*: segment nie zajmuje miejsca w skompilowanym programie.
  - Do segmentu trafiają zadeklarowane (i zdefiniowane – nie `extern`) zmienne bez jawnej inicjalizacji, np.:

```
static int i;
int tab[100000]; // globalna zmienna
```

- Segment stosu
  - W starszych architekturach sprzętowych segmenty stosu i sterty rosną w przeciwnych kierunkach i zużywają wolną pamięć. W nowszych mogą być umieszczone w dowolnym miejscu.
  - Na stosie przydzielana jest pamięć dla zmiennych automatycznych (deklarowanych wewnątrz funkcji lub bloków instrukcji)
  - Na stosie przydziela się również pamięć dla formalnych parametrów funkcji
  - Na stosie umieszcza się również wartości zwracane przez funkcje (wybrane) oraz adresy powrotu.

Przydział pamięci dla programu (4)

- Sterta
  - Sterta to potencjalnie największy obszar pamięci dostępnej dla przechowywania danych programu.
  - Zmiennych na stercie **nie deklaruje się**. Zamiast tego pamięć jest przydzielana **dynamicznie** w trakcie wykonania programu.
  - Biblioteczne funkcja `malloc()` lub `calloc()` przydziela blok pamięci o żądanym rozmiarze i zwraca do niego wskaźnik.
  - Funkcja `free()` zwalnia pamięć bloku, którego adres jest przekazany przez wskaźnik.
  - W zależności od systemu, sterta może być dzielona przez procesy i dynamicznie ładowane biblioteki. Dzielona pamięć sterty może służyć do komunikacji pomiędzy procesami.
  - W szczególnych implementacjach (systemy wbudowane) sterta może być nieobecna.

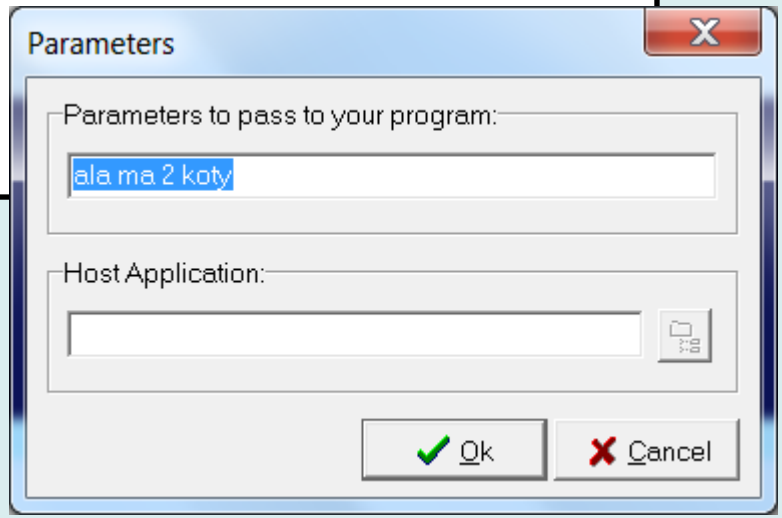
Przydział pamięci dla programu (4)

- Ostatnim blokiem są argumenty programu i zmienne środowiskowe
- Są to dane **niemodyfikowalne**!

Przykład – wypisanie argumentów wywołania

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

```
C:\lib\Dev-Cpp\Project1.exe
ala
ma
2
koty
```



Przydział pamięci dla programu (5)

- W systemie operacyjnym zdefiniowane są tzw. zmienne środowiskowe. Jest to zbiór par (*klucz*, *wartość*) gdzie nazwie zmiennej (*kluczowi*) przypisany jest tekst (*wartość*).
- Odczytane wartości są również niemodyfikowalne

```
C:\Users\pszwed>echo %TEMP%
C:\Users\pszwed\AppData\Local\Temp
```

```
int main(int argc, char*argv[])
{
  const char * varname="temp";
  char*var = getenv(varname);
  printf("%s=%s\n",varname,var);
  return 0;
}
```

```
temp=C:\Users\pszwed\AppData\Local\Temp
```

Zakres widoczności identyfikatorów (1)

Zakres widoczności (ang. *scope*) identyfikatora jest to obszar programu, w którym można się odwoływać do danego identyfikatora funkcji lub zmiennej.

Rozróżnia się następujące zakresy widoczności identyfikatorów:

- pliku
- funkcji
- bloku
- prototypu funkcji

Zakres widoczności identyfikatorów (2)

Zakres widoczności: wewnątrz pliku

- Identyfikator pojawia się poza definicją funkcji lub listą jej parametrów.
- Identyfikator jest widoczny od momentu jego deklaracji aż do końca pliku (modułu translacji)
- Nie deklarujemy identyfikatorów globalnych w plikach nagłówkowych!

```
int a =0;

void f1() {
        a++;
        b++; // błąd, zmienna b nie została zadeklarowana
}

int b=0;

void f2() {
        a++;
        b++;
        f1(); // poprawne, funkcja f1() jest widoczna
}
```

Zakres widoczności identyfikatorów (3)

Zakres widoczności: wewnątrz funkcji

- Jedynym typem identyfikatora, którego widoczność jest ściśle związana z funkcją jest etykieta instrukcji.
- Możliwość etykietowania instrukcji jest odziedziczona po niestrukturalnych językach programowania. Etykietowanie instrukcji ma zastosowanie przy użyciu **niezalecanej** instrukcji `goto` pozwalającej na przeskok do dowolnego miejsca wewnątrz funkcji.

```
void printSqrt(double x)
{
        if(x<0) goto error;
        printf("%f",sqrt(x));
        return;
error:  printf("error");
}
```

Zakres widoczności identyfikatorów (4)

Zakres widoczności: wewnątrz bloku instrukcji

Blokiem instrukcji nazywany jest ciąg instrukcji pomiędzy nawiasami { }.

Blokiem instrukcji jest:

- definicja funkcji
- obszar pomiędzy nawiasem otwierającym i zamykającym wewnątrz funkcji.

Identyfikator, którego zakres widoczności jest ograniczony do bloku instrukcji zadeklarowany jest jako element:

- listy formalnych parametrów funkcji
- listy zmiennych zdefiniowanych na początku bloku (w C++ w dowolnym miejscu wewnątrz bloku).

Zakres widoczności identyfikatorów (4)

Właściwości

- Zmienne, których zakres widoczności ograniczają się do bloku instrukcji są zawsze zmiennymi automatycznymi.
- Przydziela się im pamięć **na stosie** w momencie wejścia do bloku .
- Ich pamięć jest zwalniana w momencie opuszczenia bloku.

```
void f(int x)
{
        int y=0;
        printf("x=%d y=%d", x, y) ;
        {
                int z=1 ;
                printf("x=%d y=%d z=%d", x, y, z) ;
        }
}
```

Zakres widoczności identyfikatorów (5)

Przesłanianie

- W momencie wykonywania bloku instrukcji widoczne są wszystkie identyfikatory zdefiniowane na wyższym poziomie (w pliku, w bloku wyższego poziomu) oraz identyfikatory zadeklarowane w danym bloku.
- Jeżeli nazwy identyfikatorów pokrywają się, wówczas deklaracje w blokach niższego poziomu przesłaniają deklaracje wyższego poziomu.

```
int x=7;
void f()
{
        printf("%d\n",x); // wypisze 7
        {
                float x=8;
                printf("%f\n",x); // wypisze 8.00000
                x++;
                printf("%f\n",x); // wypisze 9.00000
        }
        printf("%d\n",x); // wypisze 7
}
```

Zakres widoczności identyfikatorów (6)

Zakres widoczności: wewnątrz prototypu

W języku C/C++ stosuje się tzw. prototypy funkcji. Prototypy są konstrukcją umożliwiającą podanie typów funkcji, które nie są widoczne w danym miejscu programu, ale które powinny być w nim użyte.

```
/* prototyp (deklaracja) funkcji */
double distance(double, double) ; // postać 1
 // lub
double distance(double xx, double yy) ; // postać 2

/* definicja funkcji */
double distance(double x, double y)
{
        return sqrt(x*x+y*y) ;
}
```

- Zakres widoczności formalnych parametrów funkcji zdefiniowanych w prototypie kończy się w momencie deklaracji.
- Nazwy parametrów występujących w prototypie są dowolne i nie muszą się pokrywać z nazwami występującymi w definicji funkcji.

Konsolidacja (1)

Moduły

- Modułem (jednostką translacji) nazywamy plik źródłowy wraz ze wszystkimi włączonymi plikami nagłówkowymi.
- W plikach nagłówkowych umieszczamy zazwyczaj:
  - prototypy wyeksportowanych (dostępnych w innych modułach) funkcji
  - deklaracje wyeksportowanych zmiennych (`extern`)

Konsolidacja (2)

Odwołania do identyfikatorów spoza modułu

- Typową praktyką przy budowie programów w C/C++ jest rozmieszczenie kodu w pewnej liczbie plików źródłowych (od kilku do kilkuset).
- W modułach zazwyczaj korzysta się z zewnętrznych funkcji oraz zewnętrznych zmiennych.
- Przed odwołaniem się do danego identyfikatora musimy poprzez deklarację poinformować kompilator o jego atrybutach (czy jest zmienną/funkcją, jaki ma typ).
  - Aby odwołać się do funkcji umieszczonej w innym module korzystamy z prototypu funkcji.
  - Aby odwołać się do zewnętrznej zmiennej globalnej stosujemy deklarację `extern`.
  - Nie możemy odwołać się do zmiennych lokalnych zdefiniowanych wewnątrz funkcji.

Konsolidacja (3)

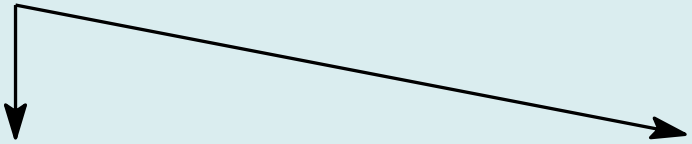
Przykład

src1.h

```
int f(double,double);
extern int v;
```

src2.h

```
int g(int a,int b);
extern int w;
```



src1.c

```
#include "src1.h"
int f(double x,double y)
{
...
}
int v;
```

src2.c

```
#include "src1.h"
#include "src2.h"
int g(int a,int b)
{
    f(1.0,2.0);
    v=4;
}
int w;
```

Konsolidacja (4)

Ukrywanie identyfikatorów wewnątrz modułu

- Standardowo, możemy odwoływać się z zewnątrz do wszystkich funkcje i zmiennych globalnych zdefiniowanych wewnątrz modułu. Jest to tzw. zewnętrzna konsolidacja (ang. *external linkage*).
- Jeżeli w dwóch różnych modułach pojawiają się definicje funkcji lub zmiennych o tej samej nazwie, wówczas podczas konsolidacji (linkowania) zostanie zasygnalizowany błąd.
- Chcąc ukryć identyfikator (wyłączyć z procesu linkowania z zewnętrznymi modułami) stosuje się modyfikator `static`. Zmienne i funkcje będą dalej widoczne wewnątrz modułu, natomiast wyłączone z globalnego procesu konsolidacji (ang. *internal linkage*).

Konsolidacja (5)

Przykład

src1.c

```
static int x;

static void g()
{
...
}
int x1;
extern int x2 ;

void f1()
{
        g(); // g() w src1.c
        x=1; // x w src1.c
        x2=x;// x2 w src2.c
}
```

src2.c

```
static int x;

static void g()
{
...
}
int x2;
extern int x1 ;

void f2()
{
        g();  // g() w src2.c
        x=1;  // x w src2.c
        x1=x; // x1 w src1.c
}
```