Wykład 14: Biblioteki programistyczne.

dr inż. Andrzej Stafiniak

Wrocław 2023

Biblioteki programistyczne

- **Biblioteki programistyczne** (*ang. libraries*) stanowią zbiór zdefiniowanych m.in. funkcji, typów danych czy stałych, które można wykorzystać w kodach źródłowych programów.
- **Biblioteki programistyczne** pozwalają na użycie tego samego fragmentu kodu realizującego zdefiniowane zadanie przez wiele różnych aplikacji.
- **Biblioteki** stanowią już **skompilowany** kod źródłowy, więc pozwalającym na skrócenie czasu kompilacji całego programu.
- Skompilowany kod biblioteki nie jest samodzielnym plikiem wykonującym program – nie posiada zdefiniowanej funkcji `main()`, ale łączy się go z kodem wykonywalnym aplikacji w celu poprawnego jej działania.
- **Biblioteki programistyczne** można podzielić na:
  - biblioteki statyczne,
  - biblioteki dynamiczne.

Biblioteki statyczne

- **Biblioteki statyczne** - (*ang. static libraries*) – biblioteki, które dołączane są do aplikacji w czasie kompilacji, na etapie **linkowania/konsolidacji** kodu wykonywalnego, w rezultacie czego kod bibliotek statycznych jest zawarty w wynikowym/ostatecznym kodzie wykonywalnym aplikacji.

Proces tworzenia pliku wykonywalnego z kodu źródłowego

Preprocesor

Preproceesing

- Usuwanie komentarzy
- Rozwijanie makr
- Dołączanie i rozwijanie plików nagłówkowych i źródłowych (*.h, *.c)

Kompilator

Kompilacja

- Pobiera dane wyjściowe z preprocesora i generuje kod w języku asemblera (specyficzny dla docelowego procesora)
- Pliki *.s, *.asm

Asemblacja

- Konwersja kodu asemblera do kodu maszynowego (binarnego)
- Pliki obiektowe *.o

Asembler

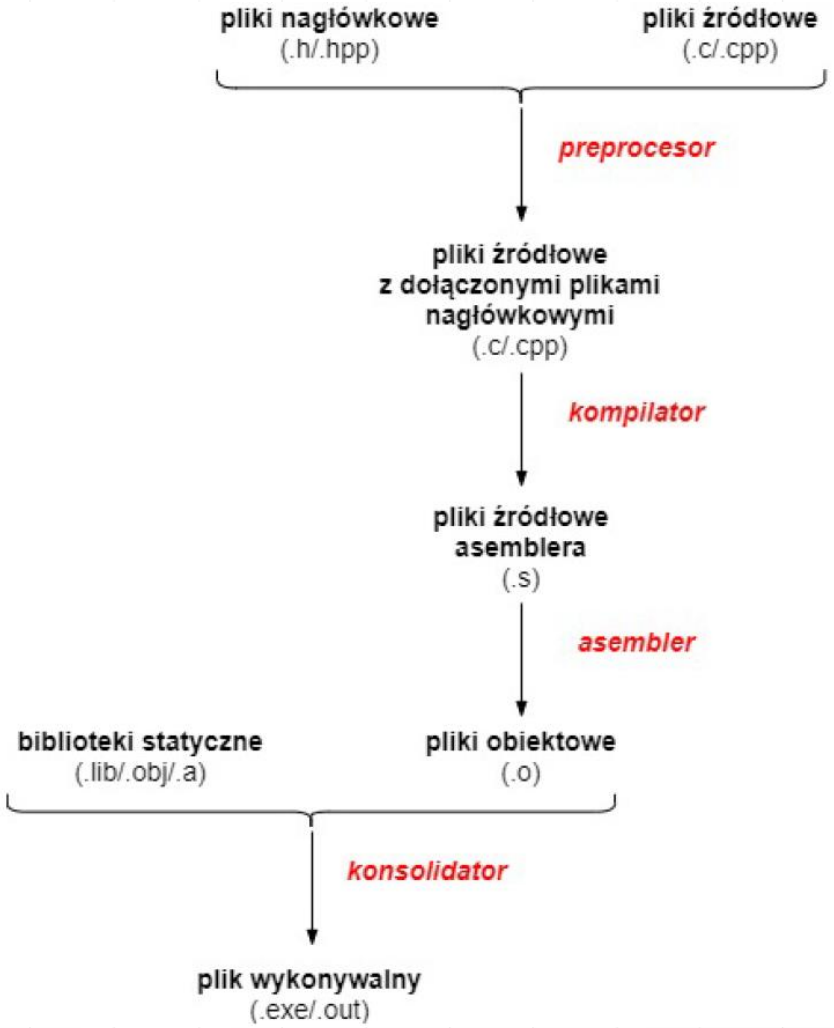
Konsolidacja (Linkowanie)

- Scalanie wszystkich plików z kodem maszynowym z różnych modułów w jeden plik wykonywalny (*.exe, *.out)

Linker

Proces tworzenia pliku wykonywalnego z kodu źródłowego

- czyli proces budowania aplikacji



Biblioteki statyczne

- **Biblioteki statyczne** - (*ang. static libraries*) – biblioteki, które dołączane są do aplikacji w czasie kompilacji, na etapie linkowania/konsolidacji kodu wykonywalnego, w rezultacie czego kod bibliotek statycznych jest zawarty w wynikowym/ostatecznym kodzie wykonywalnym aplikacji.
- Cechą charakterystyczną **nr 1** (/wadą) tego rozwiązania jest **zwiększenie rozmiaru aplikacji**, przez skopiowanie potrzebnych fragmentów kodu z biblioteki.
- Cechą charakterystyczną **nr 2** (/wadą) tego rozwiązania jest to, że jeśli wprowadzone zostaną jakieś zmiany w bibliotece po czasie **konsolidacji** kodu do **pliku wykonywalnego**, program ten będzie wymagał ponownej kompilacji w celu uaktualnienia zmian.
- W systemach z rodziny Windows **biblioteki statyczne** maja najczęściej rozszerzenie ***.lib** albo ***.obj** ,
- a w systemach z rodziny Unix rozszerzenie ***.a**.

Biblioteki dynamiczne

- **Biblioteki dynamiczne** - (*ang. dynamic-link libraries*) - biblioteki, które dołączane są do aplikacji na etapie jej uruchomienia (linkowane w czasie działania programu).
- Pojedyncza instancja biblioteki dynamicznej może być współdzielona przez wiele aplikacji jednocześnie (kod biblioteki nie jest kopiowany do kodu wykonywalnego w aplikacji).
- Aplikacje korzystające z **biblioteki dynamicznej** tworzy z nią silną zależność. Brak biblioteki we wskazanym miejscu może powodować nieprawidłowe działanie programu.
- W systemach z rodziny Windows **biblioteki dynamiczne** maja rozszerzenie ***.dll** *(ang. dynamic-link library)*,
- a w systemach z rodziny Unix rozszerzenie ***.so** (*ang. shared object*).

Biblioteki dynamiczne

- Cechą charakterystyczną **nr 1** (/zaletą) **bibliotek dynamicznych** w porównaniu do **statycznych** jest to, że plik wykonywalny aplikacji zajmuje mniej miejsca (oszczędność pamięć).
- Cechą charakterystyczną **nr 2** (/zaletą) **bibliotek dynamicznych** w porównaniu do **statycznych,** jest to że aktualizacja biblioteki nie wymaga ponownego procesu tworzenia pliku wykonywalnego aplikacji (kompilacji).
- Cechą charakterystyczną **nr 3** (/wadą) **bibliotek dynamicznych** jest zwiększony czas wykonywania programu związany z koniecznością wczytania i wykonania funkcji bibliotecznych.

Implementacja biblioteki

Aby zaimplementować **bibliotekę programistyczną** i wykorzystać ją w kodzie aplikacji należy:

- 1 - zdeklarować **interfejs** biblioteki,
- 2 - **zaimplementować funkcje** biblioteczne,
- 3 - **skompilować** kod do postaci **biblioteki statycznej** lub **biblioteki dynamicznej**,
- 4 - załączyć **interfejs** biblioteki w kodzie aplikacji,
- 5 - dołączyć bibliotekę do aplikacji na etapie **konsolidacji** kodu (biblioteki statyczne) lub **wykonania programu** (biblioteki dynamiczne).

Implementacja biblioteki

Interfejs programistyczny (*ang. application programming interface, API*) określa w jaki sposób zachodzi komunikacja miedzy programami, a w przypadku **interfejsu bibliotek**, miedzy programem a biblioteką.

Interfejs biblioteki stanowi plik nagłówkowy z deklaracjami funkcji bibliotecznych bez ich implementacji. Każda funkcja powinna być dokładnie opisana (udokumentowana komentarzem), tak aby programista mógł zastosowywać wybraną funkcję bez analizować jej implementacji (definicji, która często jest ukryta).

Wprowadzenie **interfejsu biblioteki** umożliwia rozdzielenie deklaracji funkcji bibliotecznych od ich implementacji (kodu z definicją). Takie podejście pozwala na **zmianę działania** czy **aktualizację aplikacji** oraz tworzenie oprogramowania dla różnych systemów czy architektur, po przez podmianę implementacji wybranych funkcji, a nie zmianę interfejsu. Wiąże się to z brakiem konieczności modyfikacji kodu źródłowego oprogramowania.

Interfejs biblioteki

Interfejs biblioteki stanowi plik nagłówkowy z deklaracjami funkcji bibliotecznych.

math_functions.h

```
#pragma once

/**
 * adding two double-precision floating-point numbers
 * @param x first addent
 * @param y second addent
 * @return sum of addents x and y
 */
double add(double x, double y);

/**
 * subtracting two double-precision floating-point numbers
 * @param x minuend
 * @param y subtrahend
 * @return difference of minuend and subtrahend
 */
double subtract(double x, double y);

/**
```

```
// Function pointer type of the illegal division operation handler
typedef double (*IllegalOperationHandler)();

/**
 * dividing two double-precision floating-point numbers
 * @param x dividend
 * @param y divisor
 * @param illegalOperationHandler handler to be called on division by zero
 * @return value returned by the illegalOperationHandler invocation in case of division by zero or quotient of divident and divisor otherwise
 * The function invocation result is undefined if illegalOperationHandler is NULL
 */
double divide(double x, double y, IllegalOperationHandler illegalOperationHandler);

/**
```

Interfejs biblioteki

Interfejs biblioteki stanowi plik nagłówkowy z deklaracjami funkcji bibliotecznych.

math_functions.h

```
#pragma once

/**
 * adding two double-precision floating-point
   numbers
 * @param x first addent
 * @param y second addent
 * @return sum of addents x and y
 */
double add(double x, double y);

/**
 * subtracting two double-precision floating-point
   numbers
 * @param x minuend
 * @param y subtrahend
 * @return difference of minuend and subtrahend
 */
double subtract(double x, double y);

/**
```

Interfejs biblioteki powinien charakteryzować się:

- małym stopniem skomplikowania,
- dobrą dokumentacją (dlaczego?).

Implementacja funkcji bibliotecznych

Implementacje **funkcji bibliotecznych** zawarte są w plikach źródłowych, do których programista może nie mieć dostępu (kodu źródłowego przed kompilacją).

math_functions.c

```c
#include "math_functions.h"

double add(double x, double y) {
  return x + y;
}

double subtract(double x, double y) {
  return x - y;
}

double multiply(double x, double y) {
  return x * y;
}
```

```c
double divide(double x, double y,
   IllegalOperationHandler illegalOperationHandler)
   {
  if (y == 0)
    return illegalOperationHandler();
  else
    return x / y;
}

double modulus(double x) {
  return x < 0 ? -x : x;
}
```

Kompilacja do postaci biblioteki

W celu stworzenia pliku **biblioteki programistycznej** należy skompilować plik nagłówkowy (**interfejs biblioteki**) z plikami źródłowymi zawierającymi **implementację funkcji bibliotecznych**.

W tym celu należy skompilować pliki źródłowe do postaci **piklów obiektowych *.o**:

```
#include <plik_naglowkowy.h>
```

- biblioteka statyczna

```
gcc -c plik_zrodlowy.c
```

bez konsolidacji

- biblioteka dynamiczna

```
gcc -fPIC -c plik_zrodlowy.c
```

(position-independent code) określa, że działanie wygenerowanego kod maszynowego funkcji bibliotecznych nie jest zależne od konkretnego położenia w wirtualnej przestrzeni adresowej (cecha dynamiczności).

```
plik_obiektowy.o
```

Kompilacja do postaci biblioteki

W celu stworzenia pliku **biblioteki programistycznej** należy skompilować plik nagłówkowy (**interfejs biblioteki**) z plikami źródłowymi zawierającymi **implementację funkcji bibliotecznych**.

Następnie **pliki obiektowe *.o** łączy się w jeden plik biblioteki:

- **biblioteka statyczna**

```
ar -cvru nazwa_biblioteki.lib pliki_obiektowe.o
```

Rozszerzenie plików bibliotek statycznych **`*.lib`**

Rozszerzenie plików bibliotek dynamicznych **`*.dll`**

- **biblioteka dynamiczna**

```
gcc -shared -o nazwa_biblioteki.dll pliki_obiektowe.o
```

Kompilacja do postaci biblioteki

Aby móc wykorzystać stworzone biblioteki należy w piku źródłowym z funkcją `main()` umieścić nagłówek **(interfejs)** biblioteki `"plik_naglowkowy.h"` oraz przeprowadzić kompilację pliku źródłowego `main.c` aplikacji razem z plikiem biblioteki.

```
gcc -o app.exe main.c nazwa_biblioteki.lib/.dll
```

```
gcc -o app.exe main.c nazwa_biblioteki.lib/.dll -L[sciezka]
```

Funkcje biblioteczne a pamięć programu

Fragmenty kodu **bibliotek statycznych** dodawane są do wynikowego kodu wykonywalnego opracowanego programu.

main.cpp ← Kod źródłowy aplikacji

```
#include <iostream>
#include <stdlib.h>
#include <chrono>
#include "sort.h"

int main ()
    double t
    srand(0)
    long lon

    for(long
        for

        }
        //bu
        autc
        sort
        autc
        dura

        //qu
        autc
        sort
        autc
        dura

        //me
        autc
        sort
        autc
        dura
    }

    std::cou
    std::cou
```

sort.cpp ← Kod źródłowy biblioteki

```
#include <cstring>
#include "sort.h"

namespace sorting {

namespace {

void swap(doub
    double tmp
    *a = *b;
    *b = tmp;
}

int partition(
    double piv
    unsigned i
    for (unsig
        if (ar
            sw
            pi
        }
    }
    swap(array
    return piv
}

void merge(dou
    const unsi
    double arr
    std::memcp

    int iterat
    int loIter
    int hiIter
```

sort.h ← Interfejs biblioteczny

```
#pragma once

namespace sorting {

/**
 * sort array in ascending order using bubble so
 * @param array pointer to array of double-preci
 * @param n size of array
 */
void bubbleSort(double * array, int n);

/**
 * sort array in ascending using quick sort algo
 * @param array pointer to array of double-preci
 * @param lo low index of array
 * @param hi high index of array
 */
void quickSort(double * array, int lo, int hi);

/**
 * sort array in ascending order using merge sor
 * @param array pointer to array of double-preci
 * @param lo low index of array
 * @param hi high index of array
 */
void mergeSort(double * array, int lo, int hi);

} // namespace sorting
```

Funkcje biblioteczne a pamięć programu

Fragmenty kodu **bibliotek statycznych** dodawane są do wynikowego kodu wykonywalnego opracowanego programu.

main.cpp

```
#include <iostream>
#include <stdlib.h>
#include <chrono>
#include "sort.h"

int main ()
    double t
    srand(0)
    long lon

    for(long
        for

        }
        //bu
        autc
        sort
        autc
        dura

        //qu
        autc
        sort
        autc
        dura

        //me
        autc
        sort
        autc
        dura
    }

    std::cou
    std::cou
```

sort.cpp

```
#include <cstring>
#include "sort.h"

namespace sorting {

namespace {

void swap(doub
    double tmp
    *a = *b;
    *b = tmp;
}

int partition(
    double piv
    unsigned i
    for (unsig
        if (ar
            sw
            pi
        }
    }
    swap(array
    return piv
}

void merge(dou
    const unsi
    double arr
    std::memcp

    int iterat
    int loIter
    int hiIter
```

sort.h

```
#pragma once

namespace sorting {

/**
 * sort array in ascending order using bubble so
 * @param array pointer to array of double-preci
 * @param n size of array
 */
void bubbleSort(double * array, int n);

/**
 * sort array in ascending using quick sort algo
 * @param array pointer to array of double-preci
 * @param lo low index of array
 * @param hi high index of array
 */
void quickSort(double * array, int lo, int hi);

/**
 * sort array in ascending order using merge sor
 * @param array pointer to array of double-preci
 * @param lo low index of array
 * @param hi high index of array
 */
void mergeSort(double * array, int lo, int hi);

} // namespace sorting
```

Plik źródłowy `sort.cpp` skompilowano do postaci bibliotek:

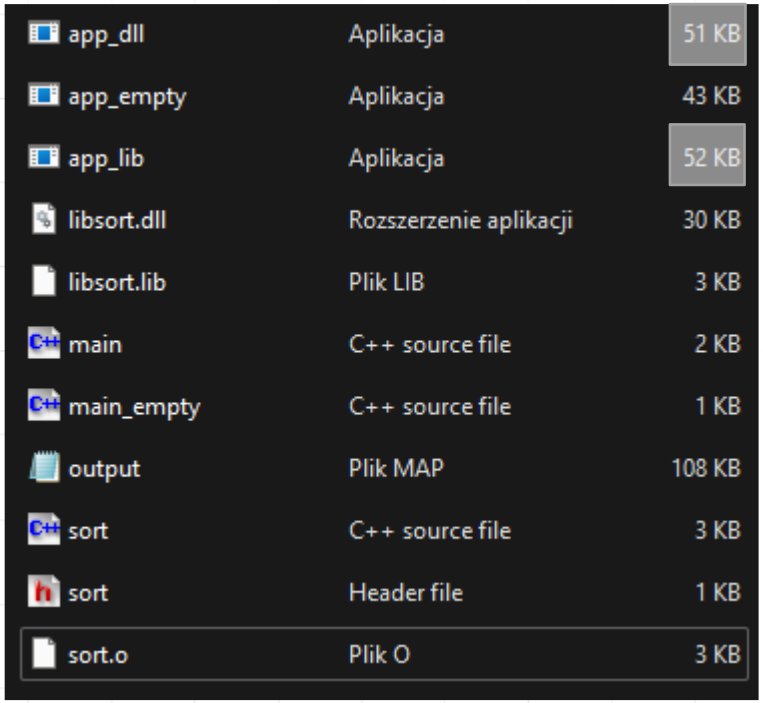
`libsort.lib, libsort.dll`

Następnie plik źródłowy `main.cpp` skompilowano razem z odpowiednią biblioteką do postaci kodu wykonywalnego aplikacji:

`app_lib.exe, app_dll.exe`

Funkcje biblioteczne a pamięć programu

Fragmenty kodu **bibliotek statycznych** dodawane są do wynikowego kodu wykonywalnego opracowanego programu.



Plik źródłowy `sort.cpp` skompilowano do postaci bibliotek:

`libsort.lib, libsort.dll`

Następnie plik źródłowy `main.cpp` skompilowano razem z odpowiednią biblioteką do postaci kodu wykonywalnego aplikacji:

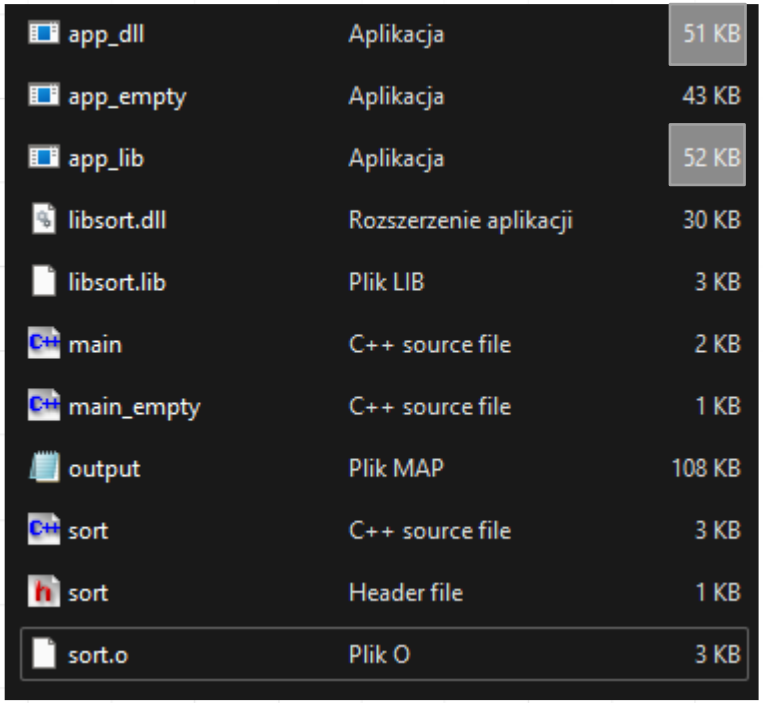
`app_lib.exe, app_dll.exe`

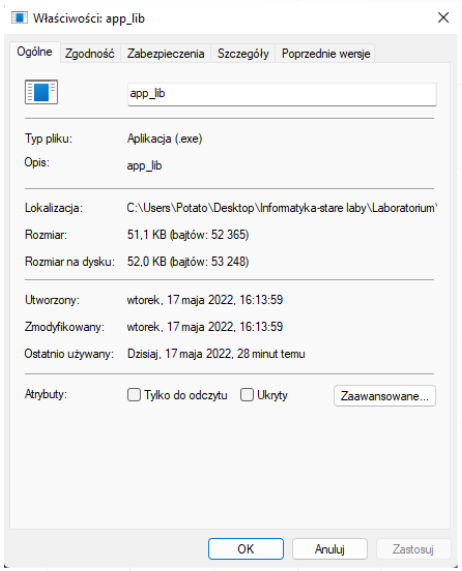
Funkcje biblioteczne a pamięć programu

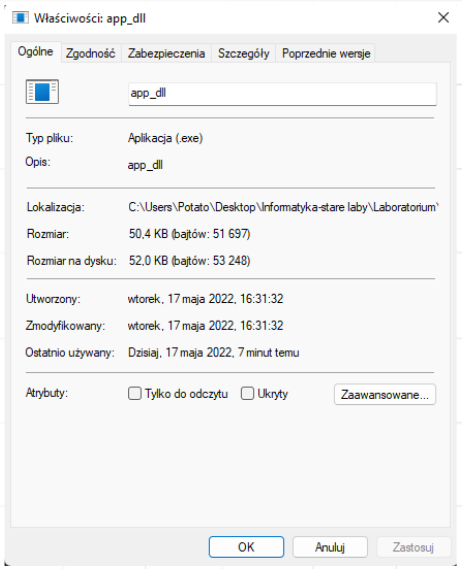
Fragmenty kodu **bibliotek statycznych** dodawane są do wynikowego kodu wykonywalnego opracowanego programu.

`libsort.lib, libsort.dll`

`app_lib.exe, app_dll.exe`

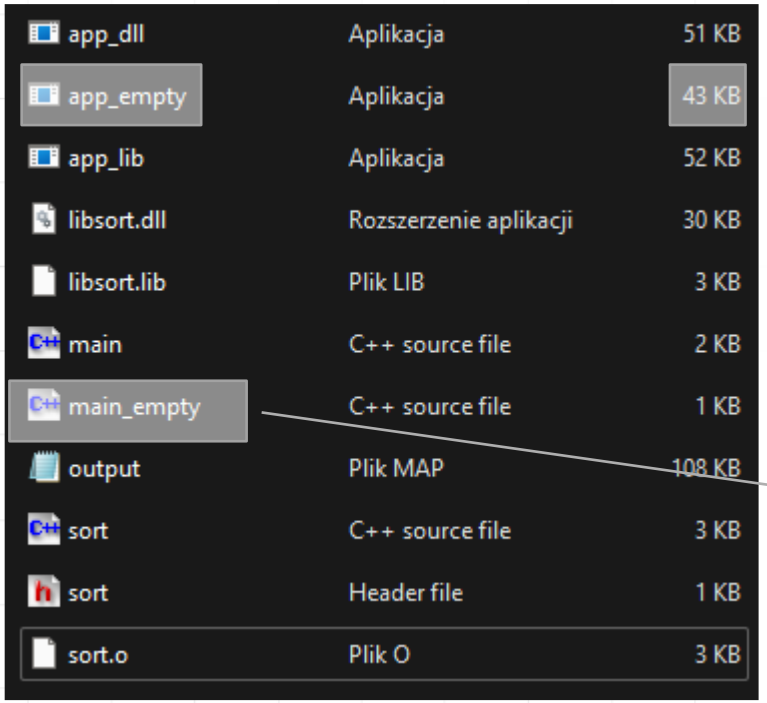






Funkcje biblioteczne a pamięć programu

Fragmenty kodu **bibliotek statycznych** dodawane są do wynikowego kodu wykonywalnego opracowanego programu.



Biblioteka standardowa jest automatycznie dołączana do programu podczas jego kompilacji.

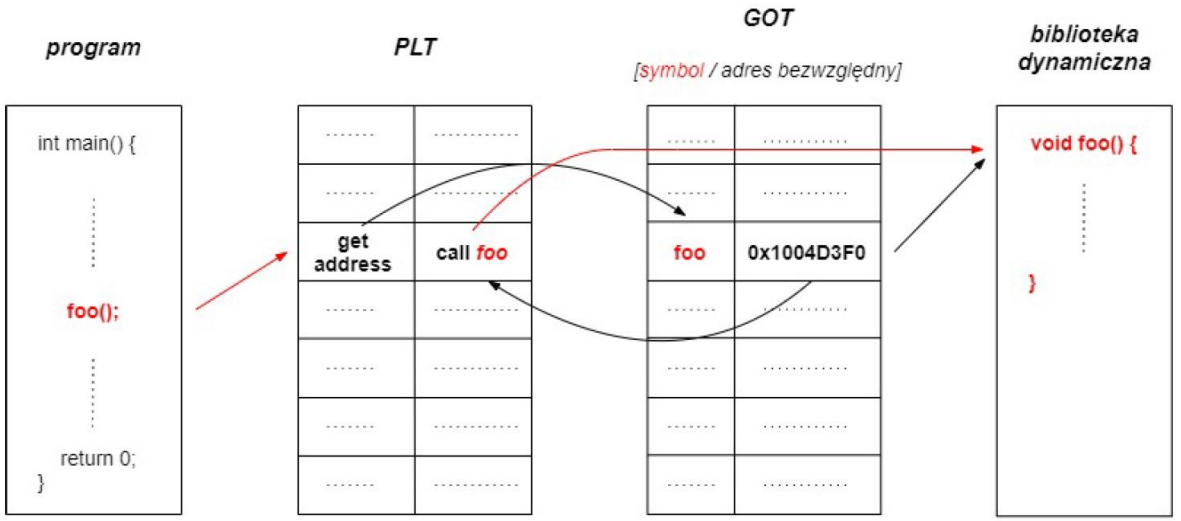
Zapewnienia obsługę urządzeń wejścia-wyjścia, podstawowych algorytmów czy struktur danych.

```
#include <iostream>

int main () {

    return 0;
}
```

Funkcje biblioteczne a pamięć programu

W wypadku **bibliotek dynamicznych** kod funkcji bibliotecznych nie jest dodawany do kodu wykonywalnego aplikacji.

Stosowana jest globalna tablica offsetów (ang. global offset table, GOT), w której mapowane są symbole funkcji bibliotecznych na ich adresy bezwzględne. Odwołanie się do funkcji biblioteki dynamicznej zachodzi z wykorzystaniem tablicy powiazań (ang. procedure linkage table, PLT). W momencie wywołania funkcji sterowanie przekazywane jest do odpowiedniej komórki tablicy PLT. Tam zachodzi odczytanie bezwzględnego adresu funkcji z tablicy GOT i wykonanie zawartych pod nim instrukcji.



Funkcje biblioteczne a pamięć programu

Aby szczegółowo przeanalizować rozkład pamięci programu można wygenerować plik z rozszerzeniem `.map` lub zastosować w konsoli znane już narzędzie `nm` (instrukcja nr 5) w celu wypisania wszystkich symboli wykorzystywanych w plikach obiektowych oraz ich adresy.

Przykładowe wywołanie w celu utworzenia pliku `.map` (pliku, który będzie zawierał konfigurację pamięci) :

```
gcc -o app_lib.exe main.c libmath.lib -Xlinker -Map=output.map
```

Funkcje biblioteczne a pamięć programu

Przykładowe wywołanie w celu utworzenia pliku `.map` (pliku, który będzie zawierał konfigurację pamięci) :

```
gcc -o app_lib.exe main.c libsort.lib -Xlinker -Map=output.map
```

W sytuacji konsolidacji **biblioteki statycznej .lib z plikiem obiektowym .o,** plik `output.map` będzie zawierał informacje o położeniu funkcji bibliotecznych w segmencie pamięci .text:

`0x3ec B = 1004 B (kod biblioteki .lib)`

```
.text           0x00401884      0x3ec libsort.lib(sort.o)
                0x00401ac3                sorting::bubbleSort(double*, int)
                0x00401b7f                sorting::quickSort(double*, int, int)
                0x00401bec                sorting::mergeSort(double*, int, int)
```

`bc B = 188 B`

Funkcje biblioteczne a pamięć programu

Przykładowe wywołanie w celu utworzenia pliku `.map` (pliku, który będzie zawierał konfigurację pamięci) :

```
gcc -o app_dll.exe main.c libsort.dll -Xlinker -Map=output.map
```

W sytuacji konsolidacji **biblioteki dynamicznej .dll,** plik `output.map` będzie zawierał (również w segmencie .text) informacje o funkcjach bibliotecznych (tylko tych faktycznie używanych):

0x8 B = 8 B (hmmm? Trzeba zerknąć co się kryje w kodzie asemblera w pliku obiektowym)

```
.text          0x00404018       0x8 d000001.o
               0x00404018              sorting::bubbleSort(double*, int)
.text          0x00404020       0x8 d000002.o
               0x00404020              sorting::mergeSort(double*, int, int)
.text          0x00404028       0x8 d000003.o
               0x00404028              sorting::quickSort(double*, int, int)
```

Funkcje biblioteczne a pamięć programu

Fragmenty kod asemblera plików obiektowych aplikacji z konsolidacją bibliotek statycznych:

188 B

```
00401ac3 <sorting::bubbleSort(double*, int)>:
  401ac3:   55                      push   %ebp
  401ac4:   89 e5                   mov    %esp,%ebp
  401ac6:   83 ec 18                sub    $0x18,%esp
  401ac9:   83 7d 08 00             cmpl   $0x0,0x8(%ebp)
  401acd:   0f 84 a9 00 00 00       je     401b7c <sorting::bubbleSo
  401ad3:   c6 45 ff 00             movb   $0x0,-0x1(%ebp)
  401ad7:   c7 45 f8 00 00 00 00    movl   $0x0,-0x8(%ebp)
  401ade:   8b 45 0c                mov    0xc(%ebp),%eax
  401ae1:   83 e8 01                sub    $0x1,%eax
  401ae4:   3b 45 f8                cmp    -0x8(%ebp),%eax
  401ae7:   0f 86 8f 00 00 00       jbe    401b7c <sorting::bubbleSo
  401aed:   c7 45 f4 00 00 00 00    movl   $0x0,-0xc(%ebp)
  401af4:   8b 45 0c                mov    0xc(%ebp),%eax
  401af7:   2b 45 f8                sub    -0x8(%ebp),%eax
  401afa:   83 e8 01                sub    $0x1,%eax
  401afd:   3b 45 f4                cmp    -0xc(%ebp),%eax
  401b00:   76 65                   jbe    401b67 <sorting::bubbleSo
  401b02:   8b 45 f4                mov    -0xc(%ebp),%eax
  401b05:   8d 14 c5 00 00 00 00    lea    0x0(,%eax,8),%edx
  401b0c:   8b 45 08                mov    0x8(%ebp),%eax
  401b0f:   01 d0                   add    %edx,%eax
  401b11:   dd 00                   fldl   (%eax)
  401b13:   8b 45 f4                mov    -0xc(%ebp),%eax
  401b16:   83 c0 01                add    $0x1,%eax
  401b19:   8d 14 c5 00 00 00 00    lea    0x0(,%eax,8),%edx
  401b20:   8b 45 08                mov    0x8(%ebp),%eax
  401b23:   01 d0                   add    %edx,%eax
  401b25:   dd 00                   fldl   (%eax)
  401b27:   d9 c9                   fxch   %st(1)
  401b29:   da e9                   fucompp
  401b2b:   df e0                   fnstsw %ax
  401b2d:   9e                      sahf
  401b2e:   76 31                   jbe    401b61 <sorting::bubbleSo
  401b30:   8b 45 f4                mov    -0xc(%ebp),%eax
  401b33:   83 c0 01                add    $0x1,%eax
  401b36:   8d 14 c5 00 00 00 00    lea    0x0(,%eax,8),%edx
  401b3d:   8b 45 08                mov    0x8(%ebp),%eax
  401b40:   01 c2                   add    %eax,%edx
  401b42:   8b 45 f4                mov    -0xc(%ebp),%eax
  401b45:   8d 0c c5 00 00 00 00    lea    0x0(,%eax,8),%ecx
  401b4c:   8b 45 08                mov    0x8(%ebp),%eax
  401b4f:   01 c8                   add    %ecx,%eax
  401b51:   89 54 24 04             mov    %edx,0x4(%esp)
  401b55:   89 04 24                mov    %eax,(%esp)
  401b58:   e8 27 fd ff ff          call   401884 <sorting::(anonym
  401b5d:   c6 45 ff 01             movb   $0x1,-0x1(%ebp)
  401b61:   83 45 f4 01             addl   $0x1,-0xc(%ebp)
  401b65:   eb 8d                   jmp    401af4 <sorting::bubbleSo
  401b67:   0f b6 45 ff             movzbl -0x1(%ebp),%eax
  401b6b:   83 f0 01                xor    $0x1,%eax
  401b6e:   84 c0                   test   %al,%al
  401b70:   75 09                   jne    401b7b <sorting::bubbleSo
  401b72:   83 45 f8 01             addl   $0x1,-0x8(%ebp)
  401b76:   e9 63 ff ff ff          jmp    401ade <sorting::bubbleSo
  401b7b:   90                      nop
  401b7c:   90                      nop
  401b7d:   c9                      leave
  401b7e:   c3                      ret
```

bibliotek dynamicznych:

8 B

```
00404018 <sorting::bubbleSort(double*, int)>:
  404018:   ff 25 bc 92 40 00       jmp    *0x4092bc
  40401e:   90                      nop
  40401f:   90                      nop
```

Funkcje biblioteczne a pamięć programu

```c
#include <stdio.h>

int main() {

    return 0;
}
```

Rozmiar: 39,6 KB (bajtów: 40 644)

```c
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <complex.h>
#include <ctype.h>
#include <errno.h>
#include <fenv.h>
#include <float.h>
#include <inttypes.h>
#include <iso646.h>
#include <limits.h>
#include <locale.h>
#include <math.h>
#include <setjmp.h>
#include <signal.h>
#include <stdalign.h>
#include <stdarg.h>
#include <stdatomic.h>
#include <stdbool.h>
#include <stddef.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdnoreturn.h>
#include <string.h>
#include <tgmath.h>
#include <time.h>
#include <wchar.h>

int main() {

    return 0;
}
```

???

Funkcje biblioteczne a pamięć programu

```c
#include <stdio.h>

int main() {

    return 0;
}
```

Rozmiar: 39,6 KB (bajtów: 40 644)

```c
#include <stdio.h>

int main() {

    printf("Halo %c", '!');
    return 0;
}
```

Rozmiar: 39,8 KB (bajtów: 40 766)

```c
#include <stdio.h>

int main() {
    int x;
    printf("Halo %c", '!');
    scanf("%d",x);
    return 0;
}
```

Rozmiar: 39,8 KB (bajtów: 40 851)

Funkcje biblioteczne a pamięć programu

```c
#include <stdio.h>

int main() {
    int x;
    printf("Halo %c", '!');
    scanf("%d",x);
    printf("Halo %c", '!');
    return 0;
}
```

???

Funkcje biblioteczne a pamięć programu

```c
#include <stdio.h>

int main() {
    int x;
    printf("Halo %c", '!');
    scanf("%d",x);
    printf("Halo %c", '!');
    return 0;
}
```

Rozmiar: 39,8 KB (bajtów: 40 851)

Dodatkowo zmienieć ustawienia kompilacji na mniejszą optymalizacje, bez opytymalizacji, sprawdzić czas wykonnia aplikacji z funkcą z biblioteki dynamiczniej i statycznej

Powrót do wykładu - Funkcja c.d.

Funkcje o nieokreślonej liczbie argumentów

Funkcje o nieokreślonej liczbie argumentów (ang. variadic functions) to funkcje, które można wywołać z dowolną liczbą argumentów, np. bez argumentów, jednym czy wieloma argumentami.

Jakie znamy **funkcje o nieokreślonej liczbie argumentów?**

Funkcje o nieokreślonej liczbie argumentów

Funkcje o nieokreślonej liczbie argumentów (ang. variadic functions) to funkcje, które można wywołać z dowolną liczbą argumentów, np. bez argumentów, jednym czy wieloma argumentami.

Jakie znamy **funkcje o nieokreślonej liczbie argumentów?**

```
int printf (const char * format , ...)
```

Funkcje o nieokreślonej liczbie argumentów

Aby móc skorzystać z zmiennej listy argumentów trzeba zastosować odpowiednie makra i typy zdeklarowane w pliku nagłówkowym `<stdarg.h>` biblioteki standardowej:

- `va_list` - to typ danych przechowujący informacje potrzebne do operowania na zmiennej liście argumentów,
- `va_start` - makro umożliwiające dostęp do zmiennej listy argumentów,
- `va_arg` – makro pobierające następny argument ze zmiennej listy argumentów,
- `va_copy` (C99) – makro wykonującej kopie zmiennej listy argumentów,
- `va_end` – makro kończące iterowanie po zmiennej liście argumentów.

Funkcje o nieokreślonej liczbie argumentów

Szablon/przykład definicji funkcji o nieokreślonej liczbie argumentów:

```
#include <stdio.h>
#include <stdarg.h>

double average ( unsigned int count , ...) {

   // Typ danych uzywany przez makra va_start , va_end, va_arg
   va_list args;

   // Przekazanie argumentu count poprzedzajacego zmienna liste argumentow
   va_start (args , count );

   // Inicjalizacja zmiennej lokalnej do zera
   double sum = 0;

   for (unsigned int i = 0; i < count ; ++i)
      // Pobranie kolejnej wartosci ze zmiennej listy argumentow (oczekiwany
      typ zmiennych to double )
      sum += va_arg (args , double);

   // Zakonczono iterowanie po zmiennej liscie argumentow
   va_end (args);

   return sum/count;
}
                        int main () {
                           const unsigned int count = 5;
                           printf ("%f", average(count , 1.3 , 2.5 , 0.7 , 3.4 , -1.6));
                           return 0;
                        }
```