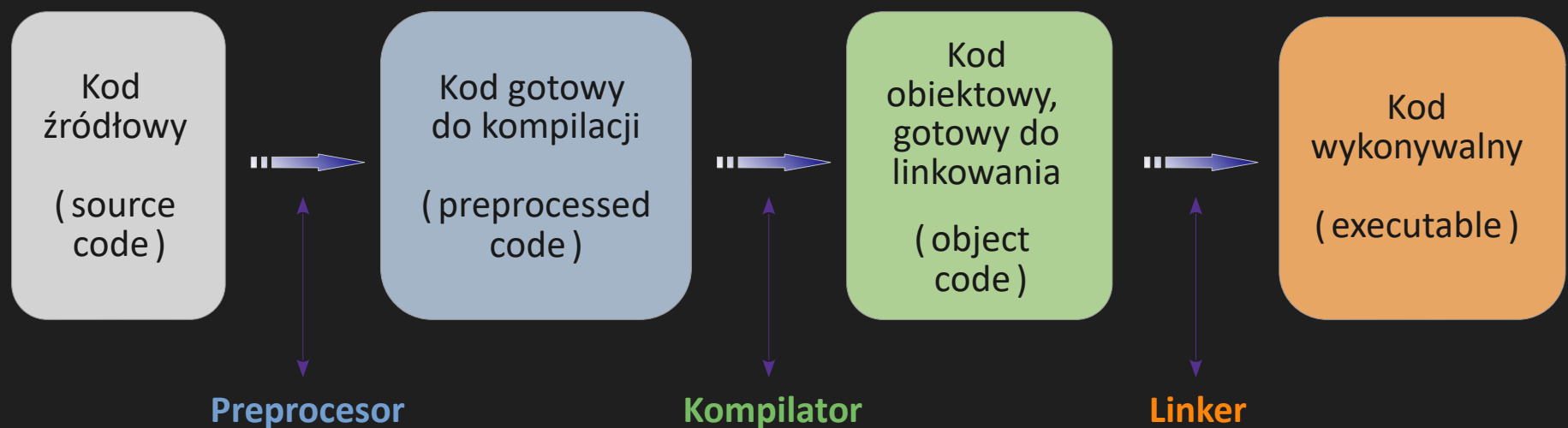


Narzędzia Wspierające Programowanie

Pliki obiektowe i linkowanie

🔗 Przypomnijmy, jak wygląda prosta droga budowania aplikacji, od kodu C/C++ do aplikacji:



- ▶ Funkcje w kodzie, skompilowane do kodu maszynowego, nazywamy obiektami. Jeśli podzielimy kompilację na etapy (`g++ -c kod.C`), to obiekty zostaną osadzone w pliku obiektowym (`.o`). Na powyższym schemacie kompilowany jest jeden kod. Ale kodów może być wiele, a funkcje wywołane w jednym pliku mogą być dostępne w innym. Zatem ostatnim etapem jest linkowanie, gdzie **linker** łączy wywołanie funkcji z jej obiektem.
- ▶ **Przykład.** Mamy klasę `complex` w plikach `complex.h` i `complex.C` oraz kod klienta `complex_client.C`, który wykorzystuje tę klasę. Skompilujemy wpierrw klasę:

```
$ g++ -c complex.C
```

↳ Dostaliśmy `complex.o`. Jest to plik z obiektami. Czym podejrzeć jego zawartość?

```
$ nm complex.C
```

 ← narzędzie wypisujące listę obiektów w plikach skompilowanych

↳ Widzimy pewne obiekty, niektóre mają w nazwach `complex` i jej metody, ale całość tonie w “dziwnych znaczkach”. To efekt tzw. **dekoracji nazwy** (“**name mangling**”).

```
$ c++filt {symbol}
```

 ← skopiujemy jeden z symboli. `c++filt` wykona „demangling” do formy pierwotnej, gdzie widzimy prototyp funkcji w kodzie C/C++

```
$ nm -C complex.o
```

 ← `nm` z opcją `-C` automatycznie „zdemangluje” obiekty na liście.

↳ Widzimy też przed nazwą tzw. “typy”. Skupmy się na dwóch:

T (t) : oznacza, że obiekt jest w tym kodzie

U : (undefined) oznacza, że symbolu w tym kodzie nie ma.

Zwykle potrzeba zewnętrznej biblioteki (pliku z obiektami, gdzie ten obiekt jest)

(Tutaj: funkcje C++ są częścią **C++ standard library** i są automatycznie łączone na etapie linkera).

● Linkowanie statyczne i dynamiczne

Istnieją dwa podejścia linkowania kodu:

- ▶ **statyczne**: utworzona aplikacja będzie mieć w sobie wszystkie potrzebne obiekty.
- ▶ **dynamiczne**: aplikacja będzie korzystać z obiektów w osobnych plikach.
Przy włączeniu aplikacji, **dynamic linker** wgra je z dysku do pamięci.

W Linuxie funkcjonuje pojęcie biblioteki (library). Ukażemy je, omawiając rodzaje linkowania.

● Linkowanie statyczne

Skompilujmy klasę `complex` do pliku obiektowego `complex.o`:

```
$ g++ -c complex.C
```

Można go teraz spakować do **biblioteki statycznej** (**static library**) o rozszerzeniu `.a`:

```
$ ar -rc libcomplex_stat.a complex.o
```

- ↳ Jest to archiwum obiektów (statycznych). Działanie to nabiera szczególnie sensu, jeśli chcemy więcej plików obiektowych połączyć w paczkę poświęconą jakiemuś zadaniu.

Aby podejrzeć zawartość, stosujemy:

```
$ ar -t libcomplex_stat.a      ← wypis plików obiektowych w bibliotece statycznej
$ nm -C libcomplex_stat.a     ← lista wszystkich obiektów w tej bibliotece
```

- ▶ Mamy kod klienta `complex_client.C` i klasę `complex`, tkwiącą w bibliotece statycznej. Zbudujmy z tego aplikację:

```
$ g++ complex_client.C -L. -lcomplex_stat -o complex_stat.exe
```

gdzie: `-L` : ścieżka poszukiwania bibliotek, `-l_____` poszuka biblioteki `lib_____`

Sprawdźmy obiekty w utworzonej aplikacji:

```
$ nm -C complex_stat.exe
```

- ↳ obiekty klasy `complex` są w kodzie. Ale nie ma obiektów standardowych C++, czyli aplikacja nie jest statyczna na 100%. Jeśli chcemy całkowicie ją “ustatyczyć”, dodajemy opcję `-static`:

```
$ g++ -static complex_client.C -L. -lcomplex_stat -o complex_fullstat.exe
```

Gdy porównamy długości tych aplikacji (`ls -ogh complex_*stat.exe`), różnica jest uderzająca. Nic dziwnego: `nm -C` pokaże nam, że wszystkie bez wyjątku obiekty tkwią w kodzie.

● Linkowanie dynamiczne

Pierwszym etapem jest utworzenie z klasy `complex` pliku “`shared library`” (`biblioteka współdzielona`), Mówi się też, że będzie zawierać `obiekty dynamiczne`.

```
$ g++ -fPIC complex.C -shared -o libcomplex.so
```

- ↪ opcja `-fPIC` (Position-Independent Code) sprawia, że adresy obiektów są względne. Zostaną ukonkretnione przy wykonaniu aplikacji, gdy dynamic loader podłączy plik `.so`.
- ↪ opcja `-shared` utworzy plik typu `.so`

Narzędzie `nm -C` pokaże nam obiekty. Ale opcja `-D` pozwoli nam skupić się na tych dynamicznych:

```
$ nm -DC libcomplex.so
```

- ↪ obiekty z symbolem `T` – to obiekty dynamiczne w naszym kodzie. Te z symbolem `U` – to obiekty dynamiczne, których nasz kod potrzebuje (liczymy, że są w innych plikach `.so`)

Zbudujmy teraz aplikację, podpinając powyższy plik `.so`:

```
$ g++ complex_client.C -L. -lcomplex -o complex_via_so.exe
```

Zajrzyjmy do obiektów (`nm -DC complex_via_so.exe`) i porównajmy długości aplikacji.

Zmieńmy na chwilę nazwę pliku `.so` i spróbujmy wykonać aplikację:

```
$ mv libcomplex.so{,.0}
$ ./complex_via_so.exe
error while loading shared library: libcomplex.so: cannot open shared object
file: No such file or directory
$ mv libcomplex.so{.0,}
```

- ↪ Widzimy, że gdy potrzebny plik `.so` zniknie, to program się nie wykona.

● Podsumujmy więc główne różnice obu metod:

Linkowanie statyczne

- kod jest niezależny od bibliotek
- ale jest dłuższy
- wykonanie: wejdzie do pamięci jako całość

Linkowanie dynamiczne

- kod jest krótszy, zadania ceduje na biblioteki
- uzależniony od bibliotek na dysku
- wykonanie: kod i biblioteki wejdą do pamięci

● Identyfikowanie potrzebnych bibliotek

Dzięki typowi `U` w wypisie z narzędzia `nm`, możemy identyfikować brakujące obiekty. Ale plik z obiektami również wie, gdzie zamierza ich poszukiwać.

```
$ objdump -p complex_via_so.exe | grep NEEDED
```

- ↳ jak widzimy, klasy `complex` aplikacja poszuka w bibliotece `libcomplex.so`. Zauważmy, że potrzebuje ona też biblioteki standardowej języka C, `libc.so`.

Z ciekawości możemy też sprawdzić, czego potrzebują pozostałe pliki obiektowe i aplikacje:

```
$ objdump -p libcomplex.so | grep NEEDED
$ objdump -p complex_stat.exe | grep NEEDED
$ objdump -p complex_fullstat.exe | grep NEEDED
```

Zauważmy, że biblioteki wypisywane są bez dokładnej lokalizacji na dysku.

Z drugiej strony, podczas wgrywania aplikacji lub biblioteki do pamięci, dynamic linker ma pomysł, gdzie poszuka tych bibliotek. Aby dowiedzieć się, gdzie planuje poszukać, piszemy:

```
$ ldd complex.so | grep NEEDED | grep libc.so | grep /lib64 | grep libc.so.6 ← miejsca poszukiwań wszystkich bibliotek
$ ldd complex.so | grep libcomplex | grep /lib64 | grep libcomplex.so ← miejsce poszukiwania danej biblioteki
```

- Przesuńmy teraz naszą bibliotekę `libcomplex.so` w inne miejsce:

```
$ mkdir lib; mv libcomplex.so lib/
```

Próba włączenia `complex_via_so.exe` się nie uda. Również `ldd` pokaże, że nie widzi biblioteki:

```
$ ldd complex.so | grep libcomplex
libcomplex.so => not found
```

Jak to naprawić? Pierwszą radą jest – zbudować aplikację na nowo, aktualizując ścieżkę dostępu do biblioteki. Ale czasem nie jest to możliwe, np. gdy nie mamy praw do kompilacji.

Linux ma zmienną środowiskową `LD_LIBRARY_PATH`. Jeśli wpisujemy tam ścieżki, to dynamic linker będzie poszukiwał bibliotek w nich. Uwaga: gdyby poszukiwane obiekty były w bibliotekach na różnych ścieżkach, to dynamic linker pobierze pierwszą pasującą, w kolejności czytania.

Ścieżkę `LD_LIBRARY_PATH` można zaktualizować chwilowo, przy wywołaniu aplikacji:

```
$ LD_LIBRARY_PATH=./lib:$LD_LIBRARY_PATH ./complex_via_so.exe
```

Można też ustawić to w sesji, a na zupełnie trwale – dopisać do `~/.bashrc` (`~/.bash_login`).

⊕ Dodatek

W środek każdej aplikacji można wstawić zmienne `RPATH` i/lub `RUNPATH`, wpisując dodatkowy katalog, na którym dynamic linker będzie poszukiwał biblioteki. W zasadzie, można to wpisać już na etapie linkowania, dodając poniższą opcję:

```
$ g++ ... -Wl,-rpath=mypath ... ← wpisze aplikacji do RUNPATH ścieżkę mypath
```

↪ (ustawi `RUNPATH`. Aby wymusić `RPATH`, dodajemy `-Wl,--disable-new-dtags`)

Jednak zwykle nie tu mamy problem. Bo jeśli możemy stanąć i zbudować kod na nowo, to zwykle zmianę ścieżki możemy uwzględnić opcją `-L`.

Raczej mamy już zbudowaną aplikację i nie możemy jej zrekompilować.

Istnieje narzędzie `patchelf`, służące wstawieniu (lub zmianie) zmiennej `R(UN)PATH` do aplikacji:

```
$ patchelf --set-rpath /pełna/sciezka/ complex_via_so.exe
```

↪ (w rzeczywistości, ustawi `RUNPATH`. Aby wymusić `RPATH`, dodajemy `--force-rpath`)

Sprawdźmy teraz, czy `ldd` widzi naszą ścieżkę. Następnie spróbujmy wywołać aplikację:

```
$ ldd complex_via_so.exe | grep libcomplex
$ ./complex_via_so.exe
```

Aby skasować ścieżkę `R(UN)PATH`, wywołujemy `patchelf` z opcją `--remove-rpath`:

```
$ patchelf --remove-rpath complex_via_so.exe
```

⊕ Dodatek

Dowiedzmy się, w jakich ścieżkach systemowych `g++` domyślnie szuka bibliotek:

```
$ g++ -print-search-dirs | grep libraries
```

Z kolei, jeśli poprzez opcję `-L_____` w poleceniu `g++` wskażemy dodatkowe ścieżki, to mają one priorytet nad ścieżkami systemowymi.

Możemy też sprawdzić “na żywo”, gdzie podczas budowania aplikacji linker szukał kolejnych bibliotek. Aby to zobaczyć, trzeba linkerowi przekazać opcję `verbose`. Np. w takiej komendzie:

```
$ g++ -Wl,--verbose complex.o -o complex.exe
```

Z racji długiego wypisu, warto go przekierować do pliku (`>myout.log`) i podejrzeć przez `less`.