(Średnio)zaawansowane programowanie w C++

Wykład 6 - zarządzanie pamięcią, łączenie C++ z C, Python, Rust

Robert Nowak

23Z

Plan wykładu

- operator `new` i `delete`,
- przydzielanie pamięci dla dużej liczby małych obiektów,
- zarządzanie nazwami,

- łączenie C i C++,
- łączenie C++ i Pythona,
- łączenie Rust z C++, Rust z Pythonem.

Operatory new i delete

- operator new
- operator new[]
- operator delete
- operator delete[]

```
//Przykłady
Foo* f = new Foo; //Przydziela pamięć, woła konstruktor
//Generuje 'bad_alloc' jeżeli brak pamięci

delete f; //woła destruktor, zwalnia pamięć

Foo* f = new Foo[N]; //Przydziela pamięć dla N elementów
//woła konstruktory
delete [] f; //woła destruktory, zwalnia pamięć

//Stary sposób zgłaszania błędów przydziału (przed ISO93)
Foo* f = new (std::nothrow) Foo; //Zwraca 0 jeżeli brak pamięci
//Uwaga!  konstruktor może rzucać wyjątek!
```

nullptr - C++11 nowe słowo kluczowe

Sposoby oznaczania pustych wskaźników

- `#define NULL (void*)0`
- `0L`
- `nullptr`

Problemy z rzutowaniem i błędy:

```
void f(char*);
void f(int);

f(0L); //niejednoznaczność
f(nullptr); //będzie wołane f(char*)
```

Obsługa błędów przydziału ::new

zanim ::new wygeneruje bad_alloc woła funkcję użytkownika

```
#include <new> //Zawiera funkcję set_new_handler

void MyNewHandler() { /* np.  zwolnienie pamięci */ }

//Ustawienie własnej funkcji obsługi niepowodzenia przydziału
set_new_handler(MyNewHandler);
//Usunięcie własnej funkcji obsługi
set_new_handler(nullptr);
```

Sensowna funkcja powinna:

- próbować „odzyskać" pamięci. Jeżeli pamięć zostanie zwolniona, to funkcja powinna zakończyć się (return).
- generować wyjątek bad_alloc
- zainstalować lub odinstalować funkcję obsługi

Definiowanie operatora new i delete dla klasy

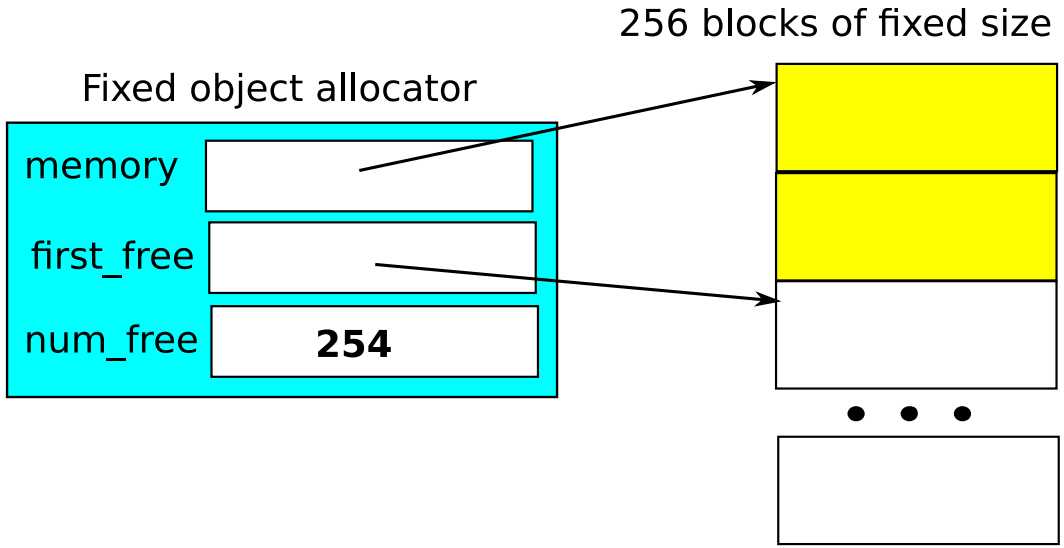
- wykrywanie błędów
- poprawa efektywności (::new jest ogólnego użytku)
- statystyki na temat wykorzystania pamięci

```
class X {
  public:
  //Własna wersja operatora new
  static void* operator new(std::size_t size) throw(std::bad_alloc){
    //np.  rejestracja
    return ::operator new(size); //woła funkcję globalną
  }
  static void //Własna wersja operatora delete
  operator delete(void* raw_memory, std::size_t size) throw ();
};
```

Uwaga! własny przydział pamięci bardzo trudny do implementacji

Przykład wykorzystania: small object allocator

- częste przydziały i zwalnianie małych obiektów o tej samej wielkości
- fragmentacja pamięci
- znaczący narzut danych pomocniczych



- **std::prm::unsynchronized_pool_resource** (C++17)
- Boost.Pool
- loki-lib.sourceforge.net

Biblioteka STL (standardowa w C++ od 1997)

Skład:

- kontenery(vector,list,...)
- iteratory
- algorytmy
- funktory
- adaptery
- alokatory

Cechy:

- wykorzystuje mechanizm szablonów
  - kod wielokrotnego użycia
  - można stosować do typów wbudowanych
- wykorzystuje wzorzec iteratora (zmniejsza liczbę algorytmów)
- bezpieczna w aplikacjach wielowątkowych
- bardzo efektywna

Biblioteka STL - alokatory

Kontenery umożliwiają dostarczenie własnej obsługi pamięci:

```
template<class T,
        class Allocator = std::allocator<T> > class vector;
```

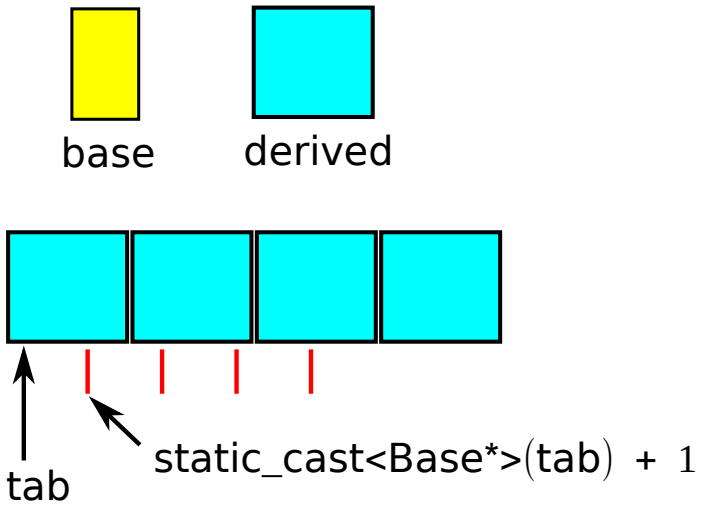
Biblioteka standardowa dostarcza kilka allokatorów

```
#include <memory_resource> //od C++17

std::pmr::unsynchronized_pool_resource //alokator dla małych obiektów

std::pmr::synchronized_pool_resource //wersja współbieżna

//zarządza dostarczonym buforem
//przy żądaniu dodatkowej pamięci zgłasza bad_alloc
std::pmr::null_memory_resource
```

Podwójna rola wskaźników

- identyfikator obiektu
- iterator dla tablicy (arytmetyka wskaźników)

```
class Base {/* ...  */};
class Derived : public Base
{/* ...  */};
void f(base* tab, int size) {
  tab[1];
};
void g(base* b);
//tablica obiektów
derived tab[N];

f(tab,N);//Niebezpieczne!
g(&tab[1]);//OK
```



Ukrywanie nazw

Nazwy „lokalne” przykrywają globalne

```
int x;
//x jest obiektem typu int
void f() {
  double x;
  //x ma typ double
}
```

Podobnie przy dziedziczeniu

```
class Base {
  int x;
  public:
  virtual void f() {
    //x jest obiektem typu int
  }
};
```

```
class Derived : public Base {
  public:
  void x(){}
  virtual void f() {
    //x jest nazwą metody
  }
};
```

Przeciążanie nazw, dyrektywa using

```
class Base {
  public:
  virtual void f();
  virtual void f(int);
}
```

```
class Derived : public Base {
  public:
  virtual void f();
}
```

```
Derived d;
d.f(2); //Błąd, nie ma takiej metody
//metoda 'f' przykrywa Base::f(), Base::f(int)!
```

```
class Base {
  public:
  virtual void f();
  virtual void f(int);
}
```

```
class Derived : public Base {
  public:
  using Base::f;
  virtual void f();
}
```

```
Derived d;
d.f(); //OK, wołana Derived::f()
d.f(2); //OK, wołana Base::f(int)
```

Łączenie C i C++

Kompilatory muszą być zgodne (w taki sam sposób reprezentować typy wbudowane)
problemy:

- dekorowanie nazw dla linkera
- struktury danych
- funkcja main
- operacje na stercie

dekorowanie nazw

linker

- w C nazwy nie mogą być przeciążane
- w C++ mogą

nazwa funkcji jest dekorowana przez kompilator C++ (name mengling, name decoration)

```
extern "C" { //Zapobiega dekorowaniu nazw
  int funkcja(int a, int b) /* ...  */
}
//Zapobiega dekorowaniu nazw jeżeli jest kompilowane przez C++
#ifdef __cplusplus
extern "C" {
#endif
  int funkcja( int a, int b) /* ...  */
#ifdef __cplusplus
}
#endif
```

funkcja main i struktury danych

Należy wybierać implementację main z C++, ponieważ

- zapewnia ona prawidłową inicjację składowych statycznych
- zapewnia wołanie destruktorów dla składowych statycznych

Struktury danych (tylko te, które są dostępne w C):

- typy wbudowane, POD
- struktury, które nie mają funkcji wirtualnych

- obiekty, które zostały powołane przez `new` powinny być zwalniane przez `delete`
- pamięć alokowana przez `malloc` powinna być zwalniana przez `free`

Łączenie C++ i Pythona

Python

- interpretowany, interaktywny język programowania
- Python Software Fundation, `www.python.org`
- programowanie **funkcyjne**, obiektowe i strukturalne
- dynamiczna kontrola typów
- brak enkapsulacji
- zarządzanie pamięcią przez garbage collection
- dokumentacja w kodzie źródłowym
- zmienna liczba argumentów funkcji i metod
- zaznaczanie bloków przez wcięcia

Potrzeba użycia różnych języków w aplikacjach

(patrz wykład 1)
System komputerowy zawsze:

- ma ograniczenia czasowe, więc tworzenie całości powinno być możliwie szybkie
- posiada pewne elementy, które są „wąskim gardłem” - powinny być zaimplementowane wydajnie (20% kodu)

System komputerowy często:

- posiada pewne elementy, których autor nie chce udostępniać (kod ukryty przed użytkownikiem)
- posiada pewne fragmenty, które powinny być dostępne dla użytkownika (aby dostosować aplikację do indywidualnych potrzeb, np. konfiguracja)

Łączenie C++ i Pythona

- powody rozszerzania Pythona w C++
  - kod wykonywany jest szybciej
  - wykorzystanie istniejącego kodu
  - kod jest ukryty przed użytkownikiem
- powody osadzania Pythona w C++
  - brak potrzeby kompilacji
  - wykorzystanie bibliotek Pythona
  - kod dostępny dla użytkownika

Interfejsy:

- Python C API
- Boost Python

Przykłady

Rozszerzanie Pythona w C++:

- utworzenie biblioteki dzielonej zawierającej funkcje w C++
- import biblioteki do Pythona i wykonanie

Przykład: hello world

Osadzanie Pythona w C++ (Boost Python):

Przykład: embedding

Łączenie Rust i C++

Wołanie kodu C++ wewnątrz Rust, kod jest 'unsafe':

- `cty` - typy zgodne z C i C++
  `c_char`, `c_uchar`
  `c_short`, `c_ushort`, `c_int`, ..., `c_ulongong`
  `int8_t`, `uint8_t`, ..., `uint64_t`
  `c_float`, `c_double`

```
extern "C" { //plik foo.h
  int funkcja(int a, int b) /* ...  */
}
//plik foo.rs
#[repr(C)]
extern "C" {
  pub fn funkcja( a: cty::c_int, b: cty::c_int) -> cty::c_int;
}
```

Następnie kompilujemy kod w C/C++ (np. do .a lub .lib).

Łączenie Python i Rust

Najczęściej rozszerzanie Pythona w Rust (tworzenie kodu w Rust, który jest wołany w Python) ponieważ:

- wydajność (kod kompilowany),
- współbieżność.

Dlaczego Rust, nie C++?

- gwarancja poprawnego zarządzania pamięcią,
- gwarancja braku wyścigów.

Narzędzia:

- `PyO3` - biblioteka Rust, pozwala tworzyć kod wołany z Pythona,
- `Maturin` - pakiet Pythona (w pip) upraszczający używanie modułów w Rust tworzonych przez `PyO3`

Dziękuję