SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Alokatori u standardnoj biblioteci C++-a

David Kerman
Voditelj: *prof.dr.sc. Siniša Šegvić*

Zagreb, lipanj 2022.

SADRŽAJ

1. Uvod

Alokatori standardne biblioteke C++ najčešće su zapostavljen dio u procesu učenja C++-a. Razlog leži u tome što ih se rijetko koristi eksplicitno, odnosno što se direktna konstrukcija objekata alokatora događa rijetko. Jedina moguća pojava termina alokator vidljiva je tijekom korištenja standardnih spremnika u C++ (primjerice vektora, mapa, listi, itd.) i to kao zadnjeg parametra.

U ovom radu bit će objašnjena dinamička memorija, memorijsko poravnjanje (engl. Alignment) kao vrlo važan koncept za baratanje dinamičkom memorijom te alokacija dinamičke memorije. Nastavno na to bit će objašnjena motivacija i primjena alokatora, definicija u standardnoj bibiloteci C++-a te programski primjer alokatora kakav je implementiran u standardnoj biblioteci. Za kraj bit će predstavljen koncept polimorfnih alokatora u kontekstu objektno orijentiranog oblikovanja te će biti objašnjen odnos polimorfnih i standardnih alokatora.

2. Dinamička memorija

Program tijekom svog izvršavanja koristi memoriju programa za pohranjivanje koda, statičkih podataka, podataka na stogu te podataka na gomili. Ovaj rad posebno razmatra dinamički alociranu memoriju koja se pohranjuje na gomili. Prije razmatranja dinamičkog zauzimanja memorije, potrebno je nešto više reći o memorijskom poravnanju.

2.1. Memorijsko poravnanje (engl. *alignment*)

Memorijsko poravnanje određuje valjane početne adrese podatkovnih regija u radnoj memoriji. Poravnanje tipično provodimo nadopunjavanjem (engl. padding) u statički i dinamički zauzetim dijelovima memorije. Sljedeći primjer ilustira poravnanje između podataka tipa char i double. Ispis programa je -8, što odgovara tome da je podatak tipa double udaljen 8 bajtova od početne adrese tipa char, čija je inicijalna veličina 1 bajt. Između tih podataka je umetnuto 7 neiskorištenih bajtova kako bi se sačuvalo poravnanje.

```
#include <stdio.h>

char c;
double d;

int main(){
   printf("%d\n", (char*)&d - &c);
}
```

Prednost memorijskog poravnanja je višestruko povećanje u performansama. Adresa podataka mora biti na adresi koja je višekratnik bajtova poravnanja. Kod primitivnih tipova poravnanje je veličina primitivnog tipa, dok je za korisničko definirane tipove poravnanje jednako poravnanju najvećeg od korištenih tipova (max. 8 bajtova na 64 bitnoj arhitekturi). [5]

```
podatak1 = 12
podatak2 = 4567
mem:
0x00 [12][45]
0x02 [67][--]
```

Primjer iznad prikazuje prvi podatak tipa byte, te drugi podatak tipa short. Byte zauzima 1 bajt, a short 2 bajta. Podaci u memoriji su spremljeni slijedno bez popune. Recimo da procesor tijekom dohvaćanja s jedne adrese troši jedan takt signala vremenskog vođenja (engl.*clock*). Tijekom dohvaćanja broja 4567 procesor bi u ovom slučaju morao utrošiti dvije jedinice vremena jer bi čitao s dvije adrese.

```
mem:
0x00 [12][--]
0x02 [45][67]
```

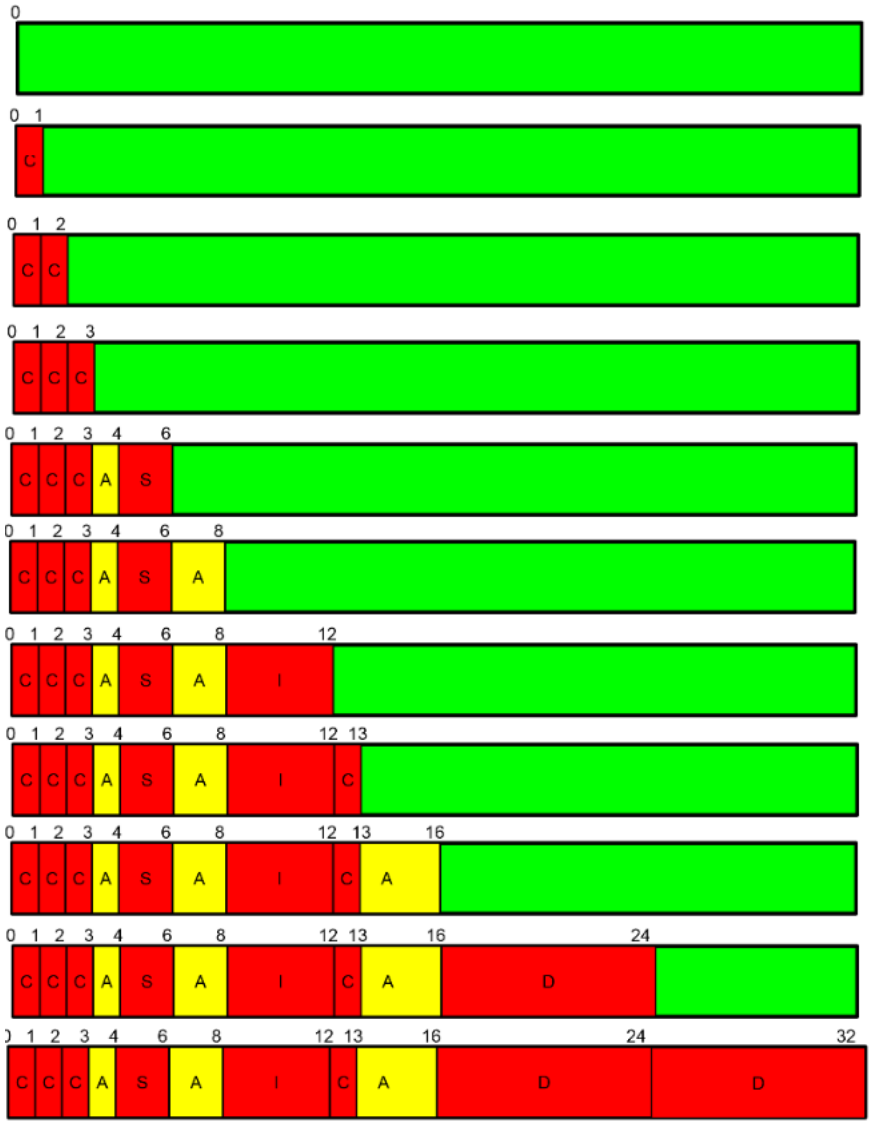
Poboljšana verzija nalazi se iznad, a prikazuje memoriju uz korišteno poravnanje. U ovom slučaju procesor koristi jednu jedinicu vremena, što je dvostruko ubrzanje. Mana ovog pristupa što se zauzima memorija koja se ne koristi. [6]

2.2. Primjer alokacije memorije

Pretpostavimo da program treba alocirati memoriju za sljedeće elemente redom: char, char, char, short, int, char, double i double. Char u memoriji zauzima 1 bajt, short 2 bajta, int 4 bajta, a double 8 bajta.

Na slici 2.1. prikazana je alokacija memorije te je vidljivo da nakon alokacije memorije za 3 char-a dolazi popuna od 1 bajta. Razlog je u tome što sljedeći na redu short ima poravnanje od 2 bajta te adresa mora biti višekratnik broja 2. Ista logika korištena je i za ostale dvije popune. Nakon što je izvršena alokacija za sve elemente vidljiva je interna fragmentacija koja je označena žutom bojom.
Memorijsko poravnanje koristi se i prilikom određivanja binarnog rasporeda strukturiranih podataka. Sljedeći primjer ilustrira takvo poravnanje. Podaci se u tom slučaju poravnavaju na višekratnik najvećeg atributnog objekta ili primitivnog tipa.

```
struct foo {
  bool b;     //velicina = 1B
              //popuna = 3B
  int i;      //velicina = 4B
};            //velicina = 8B
```



Slika 2.1: Primjer alokacije memorije

3. Alokatori

3.1. Primjena i definicija alokatora u STL-u

Alokatori su originalno bili predstavljeni kao dio STL-a koji rješava problem drugačijih tipova pokazivača. Sada služe tehničkim rješenjima kao baza, neki od memorijskih koncepata koji ih koriste su shared memory i garbage collection, a koriste ih na uniforman način bez potrebe za mijenjanjem sučelja. [2] Alokatori su predlošci razreda koji nude sučelje za alociranje i dealociranje objekata. Kolekcije STL-a mogu biti parametrizirane korisničkim alokatorima i tako prilagoditi način pohrane njihovih elemenata u memoriji računala.

Standard propisuje da alokator treba biti parametriziran elementom kolekcije T (value_type), te definirati nekoliko standardnih tipova i funkcija. Primjerice, potrebno je definirati tip size_type kojim ćemo zadavati broj alociranih objekata (tipično, to je unsigned long), tip difference_type koji će opisivati razliku pokazivača na elemente slijedne kolekcije i još neke druge tipove.

Glavni elementi alokatora su njegove dvije funkcije allocate i deallocate. Funkcija allocate alocira memoriju za N objekata tipa T, ali ne konstruira objekte, a deallocate dealocira N objekta tipa T s time da objekti moraju biti uništeni eksplicitnim pozivom destruktora.

Kao što je i rečeno, alokatori ne provode konstrukciju i destrukciju objekata. Primjer koji to ilustrira nalazi se u nastavku. Gornji dio prikazuje korištenje ključnih riječi new i delete koje onda kompajler interpretira kao korištenje operatora new i delete uz odvojenu konstrukciju i destrukciju. Njihova uloga je alokacija i dealokacija sirove memorije, s time da je preferirana upotreba „operatora new“ i „operatora delete“ pred upotrebom ključnih riječi new i delete.[3]

```
#include <stdio.h>
#include <assert.h>
#include <new>

struct A{
   A(){
     printf("ctor\n");
   }
   ~A(){
     printf("dtor\n");
   }
   int s;
};

int main(){
   A* pa = new A;
   delete pa;

   void* p = ::operator new(sizeof(A));
   A* a = new(p) A(); // placement new

   a->~A();
   ::operator delete(p);
}
```

3.2. Programski primjer alokatora

U implementaciji je korišten specifikator *constexpr*, koji specificira da se poziv funkcije može pojaviti u konstantnim izrazima. Ovaj specifikator deklarira da je moguće evaluirati vrijednost funkcije ili varijable tijekom prevođenja.

Kako bi implementirali alokator, definirat ćemo razredni predložak allocator. Predložak će imati dvije metode: allocate i deallocate. Također deklarirat ćemo tipove value_type i size_type, čije je značenje objašnjeno u prethodnom odjeljku, pomoću ključne riječi *using*. Ključna riječ *using* preferirana je od *typedef* u predlošcima, no semantika je ista. Definiran je i prazan konstruktor s operatorom noexcept, čime se specificira da se nema potencijalnih iznimaka u konstruktoru. Kod je prikazan u nastavku [6]:

```
using size_t = std::size_t;

template <class Type>
class allocator{
   public:
       using value_type = Type;
       using size_type = size_t;

       constexpr allocator() noexcept{}

       void deallocate(Type * ptr, size_t const count);
       Type* allocate(size_t const count);
};
```

Jedini argument metode allocator::allocate označava broj objekata koji moraju stati u memorijskom prostoru kojeg metoda zauzima. U izvedbi metode koristit ćemo pomoćnu funkciju i jednu pomoćnu konstantu:

- GetSizeOfBlock – funkcija vraća veličinu potrebnog memorijskog spremnika. Veličina se izračunava tako što se pomnoži broj objekata i veličina tipa objekta. Istovremeno se provjerava eventualan preljev pri množenju tako što se računa maksimalan broj objekata te to uspoređuje s parametrom poziva funkcije *count*.
- NewAlignOf - konstanta koja se izračunava u trenutku prevođenja, a odgovara memorijskom poravnanju objekta tipa Type.

```
using size_t = std::size_t;

template <size_t TypeSize>
constexpr size_t GetSizeOfBlock(size_t const count){
    constexpr bool overflowPossible = TypeSize > 1;

    if constexpr(overflowPossible){

          constexpr size_t maxPossible =
             static_cast<size_t>(-1) / TypeSize;
          if(count > maxPossible){
             throw std::bad_alloc();
          }
          return count * TypeSize;
    }
}
```

```
  template <class Type>
  inline constexpr size_t NewAlignOf = alignof(Type);
```

Sljedeće, metoda allocator::allocate poziva operator new nad bajtovima za alokaciju. Također, obavlja konverziju iz void pokazivača u pokazivač tipa Type i vraća ga kao povratnu vrijednost funkcije.

```
Type* allocate(size_t const count){
     size_t const bytes = GetSizeOfBlock<sizeof(Type)>(count);
     size_t const align = NewAlignOf<Type>;
     if (bytes != 0){
          std::cout<<"Alociram memoriju!"<<std::endl;
          void* p = ::operator new(bytes);
          return static_cast<Type*> (p);
     }
     return nullptr;
}
```

Metoda allocator::deallocate uzima void pokazivač i broj bajtova koji će biti izbrisani, te briše uz uz pomoć operatora delete.

```
void deallocate(Type * ptr, size_t const count){
     size_t const bytes = GetSizeOfBlock<sizeof(Type)>(count);
     size_t const align = NewAlignOf<Type>;

     std::cout<<"Dealociram memoriju!"<<std::endl;
     ::operator delete(ptr, bytes);
}
```

Kako bi testirali implementaciju priložen je i ispitni program koji instancira vektor koristeći implementirani alokator. U vektor se ubacuju nule, te se na kraju brišu svi elementi iz vektora. Kako bi se uvjerili u alokaciju i dealokaciju dodani su ispisi u metode allocate i deallocate.

```
#include <iostream>
#include <vector>
int main(){
     std::vector<int, allocator<int> > v(5);

     for(int i = 0; i < 5; i++) v[i] = 0;
     v.clear();
}
```

4. Polimorfni alokatori

Glavna ideja polimorfnih alokatora, koji su predstavljeni u C++17, je poboljšati standardne alokatore implementirane na bazi statičkog polimorfizma, odnosno predložaka. Polimorfne alokatore puno je lakše koristiti u odnosu na standardne jer čuvaju tip spremnika neovisno o korištenju različitih alokatora. Uzmimo na primjer $std :: vector$ sa specifičnim memorijskim alokatorom, moguće je koristiti alokator kao parametar predloška:

```
auto my_vector = std::vector<int, my_allocator>();
```

No postoji problem, ovakav vektor nije istog tipa kao vektor s drugačijim alokatorom, uključujući preddefiran (engl. default) alokator. Primjerice, takva dva vektora s drugačijim alokatorima nije moguće pridijeliti istoj varijabli:

```
auto my_vector = std::vector<int, my_allocator<int>>();
auto my_vector2 = std::vector<int, other_allocator<int>>();
auto vec = my_vector; // ok
vec = my_vector2; // pogreska
```

Specifične podatkovne strukture i tipove koje koriste polimorfni alokatori definirani su u prostoru imena *std::pmr*. U tom prostoru imena se nalaze i specijalizacije predložaka standardnih spremnika koji mogu raditi s polimorfnim alokatorima. U okviru rada su korišteni *std::pmr::vector* i *std::pmr::memory_resource*.

Polimorfni alokatori sadrže pokazivač na sučelje $memory\_resource$ te koriste dinamički polimorfizam. Kako bismo promijenili strategiju rada s memorijom, potrebno je samo zamijeniti instancu $memory\_resource$, zadržavajući tip alokatora. Ovo je dakako moguće i tijekom izvođenja. U svim drugim slučajevima polimorfni alokatori se ponašaju isto kao i standardni. Primjer dviju strategija rada s memorijom u polimorfnim alokatorima dan je kodom u nastavku. Implementacija *memory_resource*-a nije se razmatrala, no ona je slična implementaciji standardnih alokatora s metodama allocate i deallocate.

```
//definiranje memorijskog ponasanja pomocu svoje definicije
   memory_resource-a
class my_memory_resource :
    public std::pmr::memory_resource { ... };

// definicija drugog memory_resource-a
class other_memory_resource :
    public std::pmr::memory_resource { ... };

my_memory_resource mem_res;
auto my_vector = std::pmr::vector<int>(0, &mem_res);

other_memory_resource mem_res_other;
auto my_other_vector = std::pmr::vector<int>(0,
                          &mes_res_other);

auto vec = my_vector; // tip je std::pmr::vector<int>
vec = my_other_vector; // ovo je ok
// my_vector i my_other_vector imaju isti tip
```

Jedan od glavnih problema polimorfnih alokatora trenutno je što postoji nekompatibilnost novih verzija spremnika iz *std::pmr* i njima analognim u prostoru imena *std*. [4]

5. Zaključak

Alokatori iz standardne biblioteke C++-a igraju vrlo važnu ulogu u funkcioniranju raznih spremnika. U okviru rada predstavljeno je osnovno baratanje memorijom odnosno alokacija memorije. Nadalje, predstavljeni su standardni alokatori te njihova definicija i implementacija.

Također, uvedeni su polimorfni alokatori koji dolaze s određenom cijenom. Ta cijena se očituje u nekompatibilnosti sa spremnicima iz standardnog modula <*vector*> čiji su predstavnici korišteni u mnogim prijašnjim verzijama koda. No, kako sve dolazi s određenim prednostima i nedostacima, polimorfni alokatori nude dinamički polimorfizam koji se u nekim situacijma doima puno boljim od statičkog kojeg koriste alokatori putem predložaka. Jedan od primjera koji ilustira prednost dinamičkog polimorfizma je kada istoj varijabli želimo pridijeliti dva različita alokatora.

Iz sveg proizlazi da je pisanje vlastitih alokatora u nekim situacijama bolje nego puko korištenje preddefiniranih alokatora. Primjerice, u razvoju igara i igračih konzola koja nemaju puno raspoložive memorije. Na takvim sistemima važno je imati kontrolu nad svakim od manjih podsistema, te osigurati da nekritični ne kradu memoriju od kritičnog. Jedna druga primjena korištenje pool alokatora koji pomažu u smanjenju memorijske fragmentacije.[1] Ovisno o situaciji i problemu, sam programer odabire između statičkog i dinamičkog polimorfizma.

6. Literatura

[1] Compelling examples of custom c++ allocators. 2009. URL https://stackoverflow.com/questions/826569/compelling-examples-of-custom-c-allocators.

[2] Nicolai M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. 1999.

[3] Lai Shiaw San Kent. C++ standard allocator, an introduction and implementation. 2003. URL https://www.codeproject.com/Articles/4795/C-Standard-Allocator-An-Introduction-and-Implement.

[4] Alexander Klyuchev. C ++ 17 polymorphic allocators. 2022. URL https://prog.world/c-17-polymorphic-allocators/.

[5] Viacheslav Kondratiuk. C: Data structures alignment. 2013. URL https://stackoverflow.com/questions/16979791/c-data-structures-alignment.

[6] Terselich M. A guide to implement a simple c++ stl : Allocator. 2022. URL https://medium.com/@terselich/1-a-guide-to-implement-a-simple-c-stl-allocator-705ede6b60e4.