Zaawansowane CPP/Wykład 14: Zarządzanie pamięcią

From Studia Informatyczne

< Zaawansowane CPP

Spis treści

Wstęp

Dynamiczna alokacja pamięci to bardzo ważny element języka C. W C do przydziału i zwolnienia pamięci służą odpowiedno funkcje `malloc` (i jego "kuzyni") i `free`. W C++ są one również dostępne, ale używane są raczej wyrażenia `new` i `delete`. Ta zmiana ma poważną przyczynę: te wyrażenia robią więcej niż tylko przydzielanie lub zwalnianie pamięci. Wyrażenie `new` tworzy nowy obiekt, a więc nie tylko przydziela pamięc, ale również inicjalizuje go, używając odpowiedniego konstruktora. Wyrażenie `delete` niszczy obiekt, wywołując jego destruktor i dopiero potem zwalnia zajętą przez niego pamięć. W tym wykładzie pokażę, co tak naprawdę się dzieje, gdy dynamicznie tworzymy lub niszczymy obiekty.

Wyrażenia `new` i `delete` posługują się systemowymi alokatorami i dealokatorami pamięci. C++ daje nam możliwość wykorzystania w tym celu własnych implementacji. Napisanie jednak bardziej wydajnego alokatora pamięci niż alokator standardowy nie jest łatwe. Można jednak próbować zwiększyć wydajność przydzielania i zwalniania pamięci w sytuacjach szczególnych, np. jeśli używamy dużej ilości małych obiektów o stałym rozmiarze, które muszą być dynamicznie tworzone i niszczone. Pod koniec wykładu podamy prosty schemat obsługi pamięci mający zastosowanie w takiej sytuacji.

new

Przyjrzyjmy się najpierw dokładnie procesowi tworzenia pojedynczego, nowego obiektu za pomocą wyrażenia `new`:

```
X *p  = new X inicjalizator;
```

lub

```
X *p  = new (lista_argumentow) X inicjalizator;
```

Druga forma jest nazywana z przyczyn historycznych "placement new" (pochodzenie tej nazwy wyjaśnię poniżej); `inicjalizator` może być dowolnym wyrażeniem inicjalizującym, np.:

```
X *p1 = new X;
X *p2 = new X();
X x,y;
X *p3 = new X = x ;
X *p4 = new X(y);
X *p5 = new X(0);
```

Oczywiście zakładamy istnienie odpowiednich konstruktorów.

Przydział pamięci

Najpierw przydzielana jest "goła" (raw) pamięć. Służy do tego funkcja przydziału pamięci (alokator) `operator new()`:

```
void *tmp = operator new(sizeof(X));
```

lub

```
void *tmp = operator new(sizeof(X),lista_argumentow);
```

jeśli użyliśmy formy placement. Nazwa placement pochodzi od operatora `new` dostarczanego w bibliotece standardowej, który przyjmuje drugi argument typu `void *`:

```
void* operator new(std::size_t size, void* ptr) throw() {return ptr;};
```

Operator ten nie przydziela żadnej pamięci tylko zwraca wskaźnik `ptr`. Jego wywołanie nie może się nie powieść, dlatego nie rzuca żadnych wyjątków. Ta forma operatora służy do umieszczania (placement) obiektu w zadanym obszarze pamięci:

```
void *p =malloc(sizeof(X));
X *px=new (p) X;
```

stąd jego nazwa.

Środowisko C++ dostarcza jeszcze dwu wersji globalnych funkcji `operator new()`:

```
void* operator new(std::size_t size) throw(std::bad_alloc) ;
void* operator new(std::size_t size, std::nothrow_t) throw();
```

ale użytkownik może podać własne definicje, zarówno globalne, jak i dla pojedynczych klas. Odpowiednia funkcja `operator new` jest najpierw wyszukiwana w klasie `X`, a następnie w przestrzeni globalnej. Jeśli nie znajdzie się definicja odpowiadająca podanym argumentom, to wystąpi błąd kompilacji. Np. jeśli zażądamy stworzenia obiektu wyrażeniem:

```
X *p = new X;
```

to kompilator będzie szukał funkcji:

```
X::operator new(size_t);
```

a w drugiej kolejności:

```
void *tmp = ::operator new(sizeof(X));
```

Wyrażenie

```
X *p = new (3.15) X;
```

spowoduje poszukiwanie funkcji:

```
X::operator new(size_t,double);
```

lub

```
::operator new(size_t,double);
```

Pierwszy argument każdej funkcji `operator new` musi być typu `size_t` i przekazywany jest przez niego rozmiar żądanego obszaru pamięci.

Każda funkcja `operator new` zwraca `void *`. W przypadku powodzenia zwracany jest wskaźnik do przydzielonego obszaru pamięci:

```
void *p = operator new(1000);
```

W przypadku niepowodzenia `operator new` może rzucić wyjątek `std::bad_alloc` lub zwrócić wskaźnik zerowy.

Tworzenie obiektu

Jeśli przydział pamięci powiedzie się, tzn. `operator new` zwróci niezerowy wskaźnik, to następuje wywołanie konstruktora klasy `X` w celu stworzenia obiektu, który jest umieszczany w przydzielonej pamięci. Np:

```
X *p = X(1);
```

spowoduje wywołanie konstruktora:

```
X::X(int);
```

jeśli takowy istnieje. Jeśli konstrukcja się powiedzie (nie rzuci wyjątku), to proces się kończy. Jeśli jednak wywołany konstruktor rzuci wyjątek, który zostanie złapany, to wyrażenie `delete` postara się zwolnić przydzieloną pamięć w "trybie awaryjnym".

Awaryjne zwolnienie pamięci

W ramach takiej obsługi przerwania, zwalnianie pamięci przydzielonej przez `operator new` odbywa sie za pomocą odpowiadajacej mu wersji `operator delete`.

```
void operator delete(void *p,lista_argumentow) throw();
```

`operator delete` odpowiada wersji operatora `new` z taką sama listą argumentów. Jeśli lista argumentów jest niepusta, to taki operator nazywamy placement delete. Przy wywoływaniu placement delete, przekazywana mu jest lista dodatkowych argumentów, identyczna z listą dodatkowych argumentów operatora placement new, który pamięć przydzielił.

Biblioteka C++ dostarcza globalnych implementacji operatorów `delete`, odpowiadających trzem wspomnianym powyżej operatorom `new`, ale można też dodawać własne definicje, zarówno w klasie jak i w przestrzeni globalnej. Jeśli kompilator nie znajdzie żadnej odpowiedniej definicji `operator delete`, to żadna funkcja zwalniająca nie zostanie wywołana. Rozważmy następujący przykład:

```
struct X {
X(int); /* rzuca wyjątek typu int*/
void *operator new(size_t) throw(std::bad_alloc);
void *operator delete(void *p) throw();
}
```

Wyrażenie:

```
try {
X *p  = new X(1);
}
catch(int){};
```

spowoduje wywołanie:

```
void *tmp=X::operator new(sizeof(X));
X::operator delete(tmp);
```

Dodanie do klasy `X` dwu operatorów:

```
void *operator new(size_t,double) throw(std::bad_alloc);
void *operator delete(void *p,double) throw();
```

spowoduje, że wyrażenie:

```
try {
X *p  = new (3.14) X(1);
}
catch(int){};
```

wywoła:

```
void *tmp=X::operator new(sizeof(X),3.14);
X::operator delete(tmp,3.14);
```

Tę logikę zaburza trochę fakt istnienia wyróżnionej wersji funkcji `operator delete`. Są to składowe klas posiadające drugi parametr typu `size_t`:

```
void X::operator delete(void *p,size_t size);
```

Jeśli klasa nie posiada jednoargumentowego operatora `delete`, to powyższy operator jest traktowany jak jednoargumentowy (non placement). Za drugi argument podstawiany jest automatycznie rozmiar zwalnianego obiektu. Rozważmy, więc teraz taki przykład:

```
struct X {
X(int); /* rzuca wyjątek typu int*/
void *operator new(size_t) throw(std::bad_alloc);
void *operator delete(void *p,size_t) throw();
}
```

Wyrażenie:

```
try {
X *p  = new  X(1);
}
catch(int){};
```

wywoła:

```
void *tmp=X::operator new(sizeof(X));
X::operator delete(tmp,sizeof(X));
```

a wyrażenie:

```
try {
X *p  = new (3)  X(1);
}
catch(int){};
```

wywoła:

```
void *tmp=X::operator new(sizeof(X),3);
X::operator delete(tmp,3);
```

Proszę zwrócić uwagę na różnicę w wartości drugiego argumentu przekazanego do `operator delete`.

delete

Stworzony dynamicznie obiekt niszczymy wyrażeniem

```
delete p;
```

które najpierw wywołuje destruktor klasy `X`:

```
p-> ~X();
```

Jeśli to wywołanie się nie powiedzie (zostanie rzucony wyjątek) to mamy kłopot, bo nie zostanie wywołany `operator delete` w celu zwolnienia pamięci. Jest to kolejny powód aby nie rzucać wyjątków z destruktora. Poniższy programik ilustruje ten problem, doprowadzając do szybkiego wyczerpania pamięci:

```
class X {
  char a[100000];
public:
```

```
   ~X() {throw 0;}
};

main() {
   while(1){
    X *p = new X;
    try {
      delete p;
    }
    catch(int) {};
  }
}
```

Jeśli jednak nic złego się nie wydarzy, to po wywołaniu destruktora przydzielona pamięć zostaje zwolniona za pomocą funkcji `operator delete()`. O zwalnianiu pamięci dużo już napisałem przy omawianiu wyrażenia `new`. W przypadku wyrażenia `delete` dzieje się to jednak trochę inaczej. Wyrażenie `delete` używa do zwolnienia pamięci tylko funkcji `operator delete()` niebędących typu placement, tzn. posiadające jeden lub ewentualnie dwa argumenty:

```
void operator delete(void p) throw();
void operator delete(void p,size_t) throw();
```

Jest to niezależne od tego jaki `operator new` został użyty do przydzielenia pamięci. Czyli jeśli zdefiniujemy, np.:

```
struct X {
X(int); /* rzuca wyjątek typu int*/
void *operator new(size_t) throw(std::bad_alloc);
void *operator new(size_t,size_t) throw(std::bad_alloc);
void *operator new(size_t,double) throw(std::bad_alloc);
void *operator delete(void *p,size_t) throw();
void *operator delete(void *p,double) throw();
}
```

to wyrażenia:

```
X p1 = new        X;
X p2 = new (1)    X;
X p3 = new (3.14) X;
delete p3;
delete p2;
delete p1;
```

spowodują wywołanie:

```
void *tmp1 = X::operator new(sizeof(X));
void *tmp2 = X::operator new(sizeof(X),1);
void *tmp3 = X::operator new(sizeof(X),3.14);
X::operator delete(tmp3,sizeof(X));
X::operator delete(tmp2,sizeof(X));
X::operator delete(tmp1,sizeof(X));
```

operator new

Z powyższego opisu widać, że wpływ na proces dynamicznego tworzenia obiektu możemy mieć tylko poprzez własne definicje przydzielającego pamięć operatora `new`. Zanim jednak napiszemy własną wersję takiego operatora, przyjrzymy się dokładniej właściwościom standardowego operatora `new`.

Jak już wiemy funkcja operator new musi posiadać co najmniej jeden argument typu size_t. Standardowy operator new posiada tylko ten jeden argument:

```
void* operator new(std::size_t size) throw(std::bad_alloc);
```

Jeśli wszystko pójdzie dobrze, to operator new zwraca wskaźnik do obszaru pamięci o rozmiarze co najmniej size; jeśli przydział się nie powiedzie, to rzuca wyjątek std::bad_alloc.

Dokładniej rzecz biorąc operator new rzuca wyjątek tylko wtedy jeśli nie ustawiona jest funkcja obsługi błędów. Do jej ustawiania służy funkcja:

```
namespace std {
typedef void (new_handler*)();
new_hadler set_new_handler(new_handler f);
}
```

Funkcja set_new_handler ustawia nową funkcję obsługi błędów i zwraca wskaźnik do poprzedniej funkcji obsługi lub null, jeśli funkcja nie była ustawiona. Przekazanie wskaźnika null jako argumentu powoduje, że nie będzie ustawiona żadna funkcja obsługi. To co się dzieje wewnątrz operator new wygląda mniej wiecej tak:

```
while(1) {
    void *p = przydziel pamiec;
    if(proba powiodla sie) return p;
    new_handler handler = set_new_handler(0);
    set_new_handler(handler);
    if(handler)
       handler();
    else
       throw std::bad_alloc();
  }
```

Funkcja handler musi więc uzyskać więcej pamięci, rzucić wyjątek albo przerwać program. Może też ustawić inną funkcję obsługi, inaczej program będzie się wykonywał w niekończącej się pętli.

nothrow

Trzecia forma operatora new dostarczonego w bibliotece standardowej to wersja no_throw. Operator new nie musi rzucać wyjątku w razie niepowodzenia, ale musi wtedy zwrócić wskaźnik zerowy (null). Aby wywołać tę wersję operatora new korzystamy z tego, że posiada ona drugi argument typu nothrow_t.

```
void* operator new(std::size_t size,
                   const std::nothrow_t&) throw();
```

W tym celu zdefiniowana została globalna stała typu std::nothrow_t:

```
namespace std {
struct nothrow_t {};
extern const nothrow_t nothrow;
}
```

Wersję nothrow używamy więc następująco:

```
X *p=new (std::nothrow) X;
if(!p) {...};
```

operator delete

Operator `delete` musi posiadać co najmniej jeden parametr będący wskaźnikiem na zwalniany obszar pamięci:

```
void operator delete(void* ptr) throw();
```

Może to być wskaźnik zerowy, wtedy `operator new` nic nie robi. Operator `delete` nie rzuca wyjątków. Jak już opisałem to powyżej, dwuargumentowa wersja będąca składową klasy:

```
void operator delete(void* ptr) throw();
```

zachowuje się, w większości przypadków jak wersja jednoargumentowa i drugi argument zostaje automatycznie inicjalizowany rozmiarem zwalnianej pamięci.

Tablice

W powyższej dyskusji ograniczyłem się do tworzenia pojedynczych obiektów. C++ zezwala na tworzenie tablic obiektów:

```
X *px  =     new X[10];
```

Powyższe wyrażenie przydziela pamięć na 10 obiektów klasy `X` i tworzy je za pomocą konstruktorów standardowych. W przypadku niepowodzenia konstrukcji niszczy skonstruowane obiekty (jeśli takowe istnieją) i zwalnia pamięć. Alokacja gołej pamięci jest dokonywana poprzez:

```
void * operator new[] {size_t);
```

i zwalniana za pomocą:

```
void * operator new[] {size_t);
```

Przeładowywanie operatorów new i delete

Po tym przydługim, technicznym wprowadzeniu, możemy wreszcie pokusić się o napisanie własnego operatora `new` lub przeładowanie istniejącego. Do wyboru mamy wersję globalną lub funkcję składową jakiejś klasy. Globalny alokator pamięci to poważna sprawa: dotyczy działania całego programu i musi przydzielać pamięć dowolnych rozmiarów. Trudno będzie pod tym względem pobić działanie standardowej wersji `operator new`. Dlatego częściej będziemy chcieli definiować operatory `new` we własnych klasach.

Na co musimy zwrócić w takim przypadku uwagę? Mam nadzieję, że przekonałem Państwa, że do każdego operatora `new` należy dopisać odpowiednią wersję operatora `delete`, inaczej nie będziemy w stanie zapewnić bezpiecznego zachowania w sytuacji, w której zostanie rzucony wyjątek z konstruktora.

Musimy też uważać na jawne zwalnianie pamięci. Jeśli w jednej klasie zdefiniujemy kilka operatorów placement `new`, to nie będziemy mogli ich rozróżnić w poleceniu `delete`!.

Musimy też zadbać aby `operator new` albo rzucał wyjątek, albo zwracał wskaźnik pusty w razie niemożności przydziału pamięci. W innym przypadku wyrażenie `new` nie rozpozna, że przydział pamięci się nie powiódł i będzie próbować tworzyć obiekt w nieprzydzielonej pamięci.

A co z obsługą `new_handler`? W zasadzie możemy jej nie implementować i jeśli robimy to tylko na własny użytek, to pewnie nic złego się nie stanie. Ale jeśli `operator new` jest częścią zewnętrznego interfejsu klasy, to prędzej czy później któryś z użytkowników może się postarać skorzystać z `set_new_handler()`. W końcu jeżeli nazywamy naszą funkcję `operator new`, a nie np. `allocate()`, to sugerujemy, że będzie ona miała funkcjonalność `operator new`. Zwykle robimy to po to, aby skorzystać z istniejącego już kodu, który używa wyrażeń `new`. Jeśli chcemy tworzyć obiekty i przydzielać pamięć, ale nie zależy nam na interfejsie `new`, lepiej nazwać nasze alokatory inaczej.

Memory pool

Jak już sygnalizowałem na wstępie, konieczność zdefiniowania własnego operatora `new` pojawia się, gdy chcemy uzyskać wydajność lepszą niż oferowana przez standardowy `operator new`. Rozważmy więc sytuację, kiedy używamy wielu małych przedmiotów o takim samym rozmiarze. Typowy przykład to inteligentne wskaźniki. W jaki więc sposób moglibyśmy wydajnie przydzielać i zwalniać pamięć dla takich obiektów?

Jednym z prostszych sposobów jest przydzielenie za pomocą stadardowego alokatora pewnej ilości pamięci, a następnie przydzielanie z niej po kawałku pamięci na pojedyncze obiekty.

Dokładniej wygląda to tak (zob. animacja 14.1): przydzielamy obszar pamięci mogący pomieścić `N` kawałków żądanego przez nas rozmiaru. Na początku wszystkie te kawałki łączymy w listę. Ponieważ na liście bedą się znajdowały tylko kawałki nieprzydzielonej pamięci, możemy umieścić wskaźnik do następnego kawałka na liście w samym kawałku. Nasz alokator nie ma więc żadnego narzutu pamięci poza wskaźnikiem do pierwszego elementu listy (głowa).

Animacja 14.1. Działanie zasobnika pamięci.

Kiedy potrzebujemy przydzielić pamięć, to usuwamy z listy jej pierwszy element i zwracamy wskaźnik do niego. Kiedy chcemy zwolnić pamięć przyłączamy zwalniany kawałek na początku listy. Obie te operacje są bardzo szybkie. Jeśli mamy wystarczająco dużo pamięci, możemy zrezygnować ze zwalniania jej pojedynczo, tylko zwolnić cały obszar naraz, gdy nie będzie nam już potrzebny.

Kiedy będziemy potrzebowali więcej kawałków niż może ich pomieścić nasz obszar, możemy przydzielić nowy.

Napisanie klasy obsługującej taki schemat pozostawiam jako ćwiczenie. Tutaj wykorzystam gotową klasę `pool` dostarczaną w bibliotece `boost::pool`.

Ewidentnie taki schemat nie nadaje się do implementacji globalnej wersji `operator new`, która przydziela pamięć dowolnych rozmiarów. Idealnie pasuje jednak do klasowego operatora `new`. Deklarujemy więc:

```
#include<boost/pool/pool.hpp>
struct X {
  int _val;
  char c[1000];/* to tylko zwiększa rozmiar klasy*/
  static boost::pool<> pool;
public:
  X(int i=0):_val(i) {};
  operator int() {return _val;};
  void *operator new(size_t) throw(std::bad_alloc);
  void operator delete(void *) throw();
};
```

(Źródło: X_new.h)

Składowa `pool` jest składową statyczną, ponieważ musi istnieć niezależnie od obiektów klasy. Podobnie operatory `new` i `delete` automatycznie są traktowane jako metody statyczne. Ponieważ składowe statyczne klasy są inicjalizowane na zewnątrz, dodajemy do kodu linijkę:

```
boost::pool<> X::pool(sizeof(X));
```

która tworzy obiekt `X::pool` służący do przydzielania pamięci w kawałkach po `sizeof(X)` bajtów.

Następnie definiujemy `operator new`:

```
void * X::operator new(size_t size) throw(std::bad_alloc)  {
  while(1) {
    void *p = pool.malloc();
    if(p) return p;
    std::new_handler handler = std::set_new_handler(0);
    std::set_new_handler(handler);
    if(handler)
      handler();
    else
      throw std::bad_alloc();
  }
}
```

(Źródło: X_new.cpp)

Sam przydział pamięci jest najłatwiejszy: korzystamy z gotowej funkcji `malloc()` z klasy `boost::pool<>`. Reszta kodu implementuje zachowanie się operatora w przypadku braku pamięci, co funkcja `_pool.malloc()` sygnalizuje poprzez zwrócenie wskaźnika zerowego.

Operator `delete` jest dużo prostszy:

```
void X::operator delete(void *p) throw() {
  if(p)
    pool.free(p);
}
```

Alokatory

Trudno omawiać zarządzanie pamięcią w wykładzie dotyczącym programowania uogólnionego i nie wspomnieć o alokatorach STL. W poprzedniej części wykładu używałem słowa alokator na określenie każdej funkcji przydzielającej pamięć. W STL alokator jest konceptem i oznacza klasy, których obiekty służą do przydzielania pamięci dla standardowych kontenerów. Biblioteka C++ dostarcza standardową implementację szablonu alokatora, której konkretyzacje przekazywane są jako domyślny drugi lub trzeci argument szablonów kontenerów:

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

template<class T,
         class Compare = less<T>,
         class Allocator = allocator<T> >
class set;
}
```

Dlatego można używać kontenerów i nie wiedzieć nawet, że alokatory istnieją.

Wymagane elementy szablonu kontenera opiszę na przykładzie możliwej implementacji alokatora standardowego allocator.h:

```
template <class T> class      allocator {...};
```

Alokator jest szablonem przyjmujacym jako argument typ obiektów, dla których będzie przydzielał pamięć. Ponieważ parametrem szablonu kontenera jest typ, a nie szablon, można by sądzić, że alokator klasą być nie musi, bo możemy tworzyć konkretne alokatory dla konkretnych typów, np.:

```
vector<int,alokator_int> v;
```

Alokator musi jednak być szablonem, bo wewnątrz kontenera może zajść potrzeba przydzielenia pamięci dla obiektu innego typu niż typ przechowywany `T`. Dzieje się tak w przypadku kontenerów opartych na węzłach, takich jak listy czy kontenery asocjacyjne. Taki kontener musi przydzielić pamięć na węzeł, a nie na element typu `T`. Żeby móc to zrobić alokator posiada wewnętrzną strukturę:

```
template <class U>
  struct rebind { typedef pool_allocator U other; };
```

Kontener może z niej korzystać następująco:

```
typedef typename allocator<T>::rebind<node<T> >::other node_allocator;
```

Obiekty klasy `node_allocator` przydzielają pamięć na obiekty klasy `node<T>`, a nie `T`.

Alokator definiuje szereg typów stowarzyszonych:

```
public:
  typedef T                 value_type;
  typedef value_type*       pointer;
  typedef const value_type* const_pointer;
  typedef value_type&       reference;
  typedef const value_type& const_reference;
  typedef size_t            size_type;
  typedef size_t            difference_type;
```

i operator zwracający adres elementu typu `T`:

```
pointer address(reference x) const { return &x; }
  const_pointer address(const_reference x) const {
    return x;
  };
```

Miało to umożliwić używanie niestandardowych typów wskaźnikowych i referencyjnych. W praktyce te typy i funkcje muszą być zdefiniowane dokładnie tak jak powyżej (zob. S. Meyers *"STL w praktyce. 50 sposobów efektywnego wykorzystania"* oraz N.M. Josuttis *"C++ Biblioteka Standardowa. Podręcznik programisty"*).

Kontener używa obiektów klasy `allocator`, wobec czego musimy mieć możność tworzenia i niszczenia ich:

```
pool_allocator() {}
  pool_allocator(const pool_allocator&) {}
   ~pool_allocator() {}
```

Ważnym ograniczeniem narzuconym przez standard C++ jest wymaganie, aby każde dwa obiekty alokatora tej samej klasy były równoważne. Rownoważność oznacza, że pamięc przydzielona przez jeden obiekt alokatora może być zwolniona przez drugi. W naszym przypadku alokator nie posiada żadnego stanu i jego konstruktory i destruktor nic nie robią, zatem ten warunek jest spełniony. Alokator nie musi posiadać operatora przypisania, więc uniemożliwimy jego użycie:

```
private:
  void operator=(const pool_allocator&);
```

Dochodzimy wreszcie do funkcji, które zarządzają pamięcią. Funkcja:

```
pointer allocate(size_type n, const_pointer = 0) {
    return static_cast<pointer>(::operator new(n));
  };
```

przydziela pamięć dla `n` elementów typu `T`. Pamięć nie jest inicjalizowana. Proszę zwrócić uwagę, że w przeciwieństwie do `operator new` czy `malloc`, `allocate` zwraca wskaźnik `T *`, a nie `void *`. Drugi parametr funkcji `allocate` może być użyty przez bardziej wyrafinowane schematy przydziału pamięci. Tworzeniem obiektu wewnątrz przydzielonej pamięci zajmuje się funkcja

```
void construct(pointer p, const value_type& x) {
    new(p) value_type(x);
  }
```

korzystająca ze standardowego placement `new`. Funkcja:

```
void deallocate(pointer p, size_type n) throw()  {
    ::operator delete(p);
  }
```

zwalnia pamięć wskazywaną przez wskaźnik `p`. Funkcja `deallocate()` nie wywołuje destruktora. Robi to funkcja:

```
void destroy(pointer p) {
    p-> ~value_type();
  }
```

Na koniec została pomocnicza funkcja:

```
size_type max_size() const {
    return static_cast<size_type>(-1) / sizeof(T);
  }
```

która zwraca największą wartość możliwą do przekazania do funkcji `allocate`. Nie oznacza to jednak, że przydział tej pamięci musi się powieść.

Koncept alokatora wymaga jeszcze dwu operatorów testujących równoważność obiektów alokatora. Ponieważ kontenery wymagają, aby każde dwa obiekty były równoważne, te operatory zdefiniowane są następująco:

```
template <class T>
inline bool operator==(const allocator<T>&,
                       const allocator<T>&) {
  return true;
}

template <class T>
```

```
inline bool operator!=(const allocator<T>&,
                       const allocator<T>&) {
  return false;
}
```

Na koniec zabezpieczmy się jeszcze tylko na wypadek możliwości skonkretyzowania szablonu `allocator<void>` poprzez odpowiednią specjalizację:

```
template<> class allocator<void> {
  typedef void        value_type;
  typedef void*       pointer;
  typedef const void* const_pointer;

  template <class U>
  struct rebind { typedef allocator<U> other; };
};
```