(Ło)patologia C++

Omówienie i skrótowe wyjaśnienie wykładów

Maciej Nowak, R2iS2

Luty 2010

Przegląd zagadnień związanych z programowaniem obiektowym, objętych wykładami w czasie trzeciego semestru na kierunku Informatyka Stosowana.

Kontakt: mm_nowak@o2.pl

Programowanie obiektowe

Omówienie pojęć koniecznych do opanowania na zaliczenie przedmiotu

Omówienie wykładu 1: C++ wstęp – wykład ten opisuje mało ważne aspekty języka C++, takie jak jego historia, zastosowanie i cechy charakterystyczne. Żadnej wiedzy z zakresu samego kodzenia. Idziem dalej.

Omówienie wykładu 2: Powtórka z C – prócz oczywistych informacji, których posiadanie było warunkiem zaliczenia II semestru, warto zwrócić uwagę na wykaz słów kluczowych języka C++. Słowa warte uwagi to: asm, auto, const_cast, dynamic_cast, enum, explicite, extern, friend, inline, mutable, namespace, operator, register, reinterpret_cast, static, static_cast, struct, template, throw, try, typeid, typename, union, using, virtual, volatile, wchar_t

asm – wstawia instrukcję assemberową, **auto** – deklaruje zmienną automatyczną, **const_cast** – rzutowanie uzmienniające stałą, **dynamic_cast** – rzutowanie w trakcie działania programu, **enum** – definiuje typ wyliczeniowy, **explicite** – w konstruktorze zabrania domyślnej konwersji, **extern** – definiuje zmienną zewnętrzną, **friend** – deklaruje przyjaźń, **inline** – wstawia funkcję w linii, **mutable** –uzmiennia stałą, **namespace** – przestrzeń nazw, **operator** – tworzy przeładowaną funkcję operatorową, **register** – optymalizuje dostęp do zmiennej ze względu na szybkość, **reinterpret_cast** – zmienia typ zmiennej, **static** – tworzy zmienną, która istnieje przez cały czas wykonywania się programu, **static_cast** – operator rzutowania, **struct** – tworzy strukturę, **template** – tworzy szablon, **throw** – wyrzuca wyjątek, **try** – wykonuje kod który może wyrzucić wyjątek, **typeid** – opisuje typ obiektu, **typename** – w szablonach oznacza że następujący po nim symbol reprezentuje typ (synonim class), **union** – tworzy unię, **using** – używa przestrzeni nazw, **virtual** – tworzy funkcję wirtualną, **volatile** – ostrzega kompilator że zmienna może zostać nieoczekiwanie zmodyfikowana, **wchar_t** – deklaruje szeroką zmienną znajową

Inne słowa kluczowe z pewnością nie wymagają komentarza.

Jeszcze parę słów na temat wyżej wymienionych:

auto i static – globalne są static, lokalne auto, do zmiennej automatycznej należy najpierw coś zapisać, a dopiero potem czytać (nie są zerowane), a statyczne są wypełniane zerami

mutable – jeżeli metoda pewnej klasy gwarantuje, iż nie będzie modyfikować żadnej ze zmiennych tejże klasy (przydomek const) to jeśli zmienna jest opisana jako mutable to może przez taką metodę const zostać zmodyfikowana; to samo zachodzi dla całych obiektów, które są stałe

```
private:
    mutable bool done_;
public:
    void doSomething() const { ...; done_ = true; }
```

sizeof(zmienna) – podaje rozmiar, jaki zajmuje obiekt typu, którego jest zmienna

Nic więcej w wykładzie numer 2 znaleźć nie można. Reszta słów będzie omówiona wkrótce.

Omówienie wykładu 3: Powtórka z C cz. II – generalnie wykład ten przywołuje wspomnienia z II semestru. Każdy kto go zaliczył raczej nie znajdzie nic interesującego. Rzeczy, o których warto wspomnieć:

Argumenty domniemane:

`void funkcja(int liczba = 5)` – taką funkcję można wywołać bez podawania argumentu

Funkcje inline – krótkie funkcje w celu przyśpieszenia działania czyni się funkcjami inline. Wtedy kompilator zamienia każde wywołanie na kod funkcji, więc podczas wykonywania program nie traci czasu na skok do miejsca gdzie ten kod by się znajdował. Warto wiedzieć, iż w przypadku definicji klasy w zewnętrznym (względem tego z funkcją main) pliku, jeśli klasa ta ma metody inline to ich definicje muszą być w tym samym pliku.

Nazwa tablicy jest jednocześnie adresem jej początku (z pozdrowieniami dla J. Grębosza).

Wskaźnikowi przypisujemy obiekt, na który ma pokazywać używając referencji. Wskaźnik typu const nie może zmieniać obiektu, na który pokazuje. Na obiekt typu const wskazywać musi wskaźnik typu const.

`const int* const w = &iczba;` - stały wskaźnik w pokazuje na stały obiekt int

Dereferencja pozwala uzyskać wartość tego na co pokazuje wskaźnik.

Wskaźnik typu void nie przechowuje informacji o obiekcie, na którego pokazuje.

Wskaźnik do funkcji: `void (*funkcja)(argumenty)`

To tyle gwoli wykładu numer trzy.

Omówienie wykładu 4: Klasy – Podstawy używania klas w C++. Na wstępie omówienie tworzenia prostej klasy, pamiętać należy o średniku po definicji klasy. Omówienie enkapsulacji (wszystkie elementy klasy zamknięte w jednym kontenerze) i hermetyzacji (elementy klasy mogą, ale nie muszą być widoczne z zewnątrz – użycie private, public i protected). Konstruktory i destruktory na tym poziomie są trywialne i nie stanowią tematu koniecznego do omówienia.

Jeśli funkcja jest zdefiniowana wewnątrz definicji klasy to jest automatycznie typu inline.

static – czyni statycznym; w klasach zmienna statyczna istnieje i może mieć wartość nawet gdy nie istnieje żaden obiekt tejże klasy; istnieje tylko raz niezależnie od liczby powołanych obiektów, zmienna typu static może być prywatna (można ją zainicjalizować poza klasą, ale dostęp tylko z poziomu klasy)

`klasa::zmienna_statyczna;` - możliwy sposób odwołania się do element statycznego

Funkcja może być **statyczna**, służy ona wtedy do operowania na składowych zmiennych statycznych. Można ją wywołać na rzecz obiektu, ale też po prostu na rzecz klasy. Nie jest do niej wysyłany wskaźnik this. Nie jest możliwe odwołanie się do niestatecznego elementu używając funkcji statycznej. Przykładzik poniżej:

```
#include <iostream>
class CPunkt {
public:
      static int GetN(void) {  return n;  }
private:
      static int n;
};

int CPunkt::n = 5; //  nie  ma słowa static, ta linijka jest globalna
int main() {
    std::cout << "CPunkt::n = " << CPunkt::GetN() << std::endl;
}
```

Funkcje składowe typu const – zapewnia, że nie będzie modyfikowac żadnego z elementów obiektu (chyba, że te elementy są mutable). Jeśli cały obiekt jest typu const to na jego rzecz możemy wywołać funkcje tylko typu const.

`void funkcja(argumenty) const { wnętrze funkcji; }` – definicja funkcji z przydomkiem const

Uwaga! Konstruktory i destruktory nie mogą być typu const!

Funkcje zaprzyjaźnione – funkcja mająca dostęp do składowych klasy deklarującej przyjaźń, mimo iż sama nie jest funkcją składową. Funkcja może być przyjacielem więcej niż jednej klasy, ma dostęp do prywatnych, nie dostaje wskaźnika this.

```
#include <iostream>

class CPunkt {
public :
      CPunkt (float a, float b) : m_x(a), m_y(b) {} //lista inizjalizacyjna
      friend void raport(const CPunkt &pPunkt);
private:
      float  m_x ,m_y ;
};

void raport(const CPunkt &pPunkt) {
      std::cout << pPunkt.m_x << ", " << pPunkt.m_y << std::endl;
}

int main() {
      CPunkt aPunkt(1,1);
      raport(aPunkt);
}
```

Miejsca deklaracji (private, public, protected) nie ma znaczenia. Funkcja zaprzyjaźniona może być zdefiniowana wewnątrz klasy – jest wtedy typu inline, lecz jest też wciąż zwykłą funkcją nieskładową. Funkcja zaprzyjaźniona może być funkcją składową innej klasy.

Klasa zaprzyjaźniona – jeśli klasa deklaruje, że inna klasa jest jej przyjacielem, to ta zaprzyjaźniona ma dostęp do wszystkich jej funkcji składowych. Przyjaźń klas jest jednostronna, lecz nic nie stoi na przeszkodzie by ta druga klasa deklarowała przyjaźń z pierwszą.

```
#include <iostream>
class Druga; // deklaracja wyprzedzająca
class Pierwsza {
public :
    friend class Druga; // deklaracja przyjaźni
    Pierwsza(int n) : liczba(n) {}
private:
    int liczba;
};

class Druga {
public :
    void wypisz(const Pierwsza &obj) const { std::cout << obj.liczba; }
};

int main() {
    Pierwsza Pobj(5);
    Druga Dobj;
    Dobj.wypisz(Pobj);
}
```

Przyjaźń nie jest przechodnia. Jeżeli klasa A deklaruje przyjaźń z klasą B, a klasa B z klasą C to nie oznacza, że klasa A uznaję klasę C za przyjaciela

Ważne! Przyjaźń nie jest dziedziczona!

Możemy tworzyć tablicę obiektów. Należy pamiętać, iż przy liście inicjalizacyjnej podczas powoływania takiej tablicy (czyli w klamrach), przecinek oddziela inicjalizację poszczególnych obiektów, a nie składników tworzonych obiektów.

```
CPoint Tab[n] = {1, 1,  2, 2, 3, 3}; // niejawne wywołanie konstruktora
CPoint Tab[n] = {CPoint(1, 1), CPoint(2, 2), CPoint(3, 3)}; // jawne wywołanie konstruktora
```

Jeśli tworzymy tablicę obiektów używając operatora new
```
CPoint *PointTab = new CPoint[n];
```
to nasza klasa musi zawierać domyślny, bezargumentowy konstruktor.

Struktura – klasa, która przez domniemanie ma wszystkie element publiczne.

Pole bitowe – dane przechowywane na określonej ilości bitów. Tego się raczej nie używa.
```
unsigned int read : 1;
```

Unia – w wykładzie jest to najlepiej opisane. Może mieć na raz jedną wartość, za to dowolnego typu z listy podanych podczas definiowania. Rozmiar unii jest równy rozmiarowi typu z podanej listy, który zajmuje najwięcej pamięci.

```
union nazwa {int i; long l; char c;};
```

Unia bez nazwy to unia anonimowa (do składników odwołujemy się bez kropki). Po zapisaniu czegoś w unii, odczytywać należy dane tego samego typu co został zapisany. Unii używa się dla oszczędności pamięci.

To by było na tyle jeśli chodzi o wykład numer 4. Jedziemy wykład 5.

Omówienie wykładu 5: Konstruowanie obiektów klas – konstruktory, lista inicjalizacyjna, konwersje – dużo ważnych szczegółów i pełno smaczków. Yeah. Od początku…

Konstruktor: wiemy czym jest, ale czy pamiętamy, że nie może być typu static, virtual, a za to może tworzyć obiekty z przydomkiem const i volatile? Obiekty statyczne zostaną utworzone przez konstruktor na początku programu, nim zostanie wywołana funkcja main.

Lista inicjalizacyjna – pozwala nadać wartości elementom typu const, ale używając jej możemy zdefiniować także wszystkie inne (z wyjątekim statycznych) elementy klasy. Jeśli klasa posiada element będący obiektem innej klasy to można go zainicjalizować tylko używając listy (konstruktor domyślny, jeśli go nie posiada to zostanie on wygenerowany). Konstruktor obiektu będącego elementem jakiejś klasy jest wywoływany przed konstruktorem klasy, w której się znajduje. Destruktor odwrotnie. Przykład przy klasie prywatnej.

Konstruktor kopiujący – jest wywoływany z jednym argumentem będącym referencją do tej samej klasy. Dzięki niemu możemy skopiować obiekt danego typu.

```
class Klasa {
public:
      Klasa(int n) : liczba(n) {}
      Klasa(const Klasa &Kl) { liczba = Kl.liczba; }
private:
      int liczba;
};

int main()
{
    Klasa pierwsza(5);
    Klasa druga = pierwsza; // tutaj działa konstruktor kopiujący
}
```

Konstruktor kopiujący nie jest obowiązkowy, kompilator może go sam wygenerować, ale wtedy będą kłopoty gdy np. klasa zawiera elementy tworzone operatorem new. Wtedy lepiej stworzyć własny konstruktor kopiujący. Używany jest też przy zwracaniu przez funkcję obiektu danego typu.

Klasa prywatna to taka, która ma niepubliczny konstruktor. Taki konstruktor może wywołać funkcja lub klasa zaprzyjaźniona. Jeśli konstruktor jest chroniony (protected) to może go wywołać również z klasy pochodnej.

```
class Druga;
class Pierwsza {
public :
      friend class Druga;
private:
      Pierwsza(int n) : liczba(n) {}
      int liczba;
};

class Druga {
public:
      Druga(int n) : obj(n) {}
private:
      pierwsza obj;
};

int main() {
      Druga Dobj(5);
}
```

Konstruktory nazwane – jest to technika pozwalająco usprawnić pracę w sytuacji gdy istnieje dużo konstruktorów, które jak wiemy, mają zawsze tę samą nazwę – nazwę klasy. Technika „*Named Constructor Idiom*" polega na tym, że wszystkie konstruktory deklaruje się jako prywatne lub chronione, a następnie definiuje się publiczne i statyczne metody, które tworzą i zwracają nowy obiekt.

```
class Klasa {
public:
      static Klasa nazwany(int n); //  statyczna metoda zwracająca obiekt typu Klasa
private:
      Klasa(int n) : liczba(n) {}
      int liczba;
};

Klasa Klasa::nazwany(int n) {
      return Klasa(n);
}

int main() {
      Klasa Kobj = Klasa::nazwany(5);
}
```

Generalnie tworzy się po jednej statycznej metodzie dla każdego sposobu tworzenia nowego obiektu.

Agregat – klasa przechowująca jedynie dane publiczne. Nie posiada konstruktorów, funkcji, prywatnych i chronionych elementów ani funkcji wirtualnych. Nie ma również klasy nadrzędnej. Tablica, struktura też są agregatami lecz wygląda na to, iż jeżeli tablica jest tablicą obiektów, których typ posiada konstruktor to ta tablica nie jest już agregatem. Tutaj nie będę dawał przykładu. Można to sobie chyba łatwo wyobrazić.

Wskaźniki do składników klas i metod – wskaźniki mogą pokazywać bezpośrednio na elementy publiczne klasy. Mogą też pokazywać na metody. Można też zdefiniować wskaźnik, który nie pokazuje na konkretne miejsce w pamięci, ale na odległość od początku obiektu danej klasy:

```
class Klasa {
public:
    Klasa(int n) : liczba(n) {}
    int liczba;
};

int main() {
    int Klasa::*wsk = &Klasa::liczba;
}
```

Konwersja typów – przepis jak zmienić jeden obiekt w drugi. Konwersja może być jawna bądź nie. Kompilator nie umie wygenerować jej sam - musi to zrobić programista.

Niejawna konwersja jest wykonywana wszędzie tam gdzie kompilator oczekiwał obiektu jednego typu, a dostał innego (np. argument funkcji, wartość zwracana albo w obecności operatora +). Klasa, z której dokonywana jest konwersja musi dawać dostęp do koniecznych przy konwersji składników. Konstruktor nie jest dziedziczony. Konstruktor konwertujący z przydomkiem explicit gwarantuje, że nie zostanie użyty do niejawnej konwersji. Mały przykładzik na dwa typy konwersji:

```
class Liczba {
public:
      Liczba(int n) : liczba(n) {} // -pokazuje co zrobić gdy program napotka int
                                   //   w miejscu gdzie oczekiwał Liczbę
      operator int() const {       // -pokazuje co zrobić gdy program
                                   //   napotka Liczbę tam gdzie oczekiwał int
            return liczba;
      }
      int liczba;
};

int main() {
      Liczba Lobj1(5);
      Liczba Lobj2 = Lobj1 + 5; // teraz element liczba obiektu Lobj2 ma wartość 10
}
```

Przykład pokazujący niejawną konwersję z jednego typu do drugiego w przypadku gdy oba typy są zdefiniowane przez użytkownika:

```
class Pierwsza {
public:
    Pierwsza(int n) : liczba(n) {}
    int getL(void) const { return liczba; }
private:
    int liczba;
};

class Druga {
public:
    Druga(Pierwsza P) : liczba(P.getL()) {}
private:
    int liczba;
};

int main() {
    Pierwsza Pobj(5);
    Druga Dobj = Pobj; //  teraz element liczba obiektu Dobj ma wartość 5
}
```

Konwersje jawne – rzutowanie z jednego typu na drugi. C++ wprowadza kilka nowych rodzajów rzutowania na tle prostej implementacji języka C:

```
(typ na który rzutujemy)zmienna_rzutowana;
```

Nowe rodzaje:
static_cast<typ>(zmienna); - prosta metoda rzutowania
const_cast<typ>(zmienna); - rzutowanie wskaźników lub referencji różniących się modyfikatorem const; pozwala wysłać stały obiekt do funkcji, która nie gwarantuje, że nie zmieni obiektu (const)

```
#include <iostream>
class Klasa {
public:
    Klasa(int n) : liczba(n) {}
    void wypisz(Klasa &Kobj) { std::cout << Kobj.liczba << std::endl; }
private:
    int liczba;
};

int main() {
  Klasa Kobj1(5);
  const Klasa Kobj2(10);
  Kobj1.wypisz(const_cast<Klasa&>(Kobj2));
}
```

Bez ostatniej linijki ujrzymy błąd o braku dopasowania argumentu w wywołaniu funkcji wypisz()

Omówienie wykładu 6: Przeładowanie operatorów, dziedziczenie – wykład traktuje o przeładowaniu operatorów i dziedziczeniu. Należy spodziewać się dużej liczby przykładów, aczkolwiek postaram się skompresować to tak bardzo jak to tylko możliwe.

Przeładowanie operatorów umożliwia zdefiniowanie tego jak ma się zachowywać operator w obecności obiektów, które w tym przeładowaniu użyjemy (np. można ustalić, że znak plus między obiektami naszego własnego typu będzie określał sposób ich dodawania).

Można przeładować dowolny operator z wyjątkiem: **.** , **.*** , **::** , **?:**
Przeładowanie może być funkcją składową lub globalną. Nie można tworzyć własnych operatorów, zmieniać priorytetów istniejących, argumentowości (jedno czy dwuargumentowe) ani łączności.
Dla każdej klasy automatycznie generowane są operatory: **=**, **&**, **new**, **,** , **delete**

Funkcja operatorowa będąc składową przyjmuje zawsze o jeden mniej argument niż taka sama globalna. Składowa nie może być statyczna (bo w jej działaniu bierze udział wskaźnik this).
Nie może istnieć jednocześnie składowa i globalna funkcja operatorowa dla tych samych argumentów.

```
class Klasa {
public:
	Klasa(int n) : liczba(n) {}
	Klasa operator+(Klasa &Kobj) { // teraz możemy dodawać obiekty Klasa
		return liczba+Kobj.liczba;
	}
private:
	int liczba;
};

int main() {
	Klasa Kobj1(5), Kobj2(3);
	Klasa Kobj = Kobj1 + Kobj2;
}
```

To samo w wersji, w której przeładowany operator nie jest funkcją składową, a globalną:

```
class Klasa {
public:
	Klasa(int n) : liczba(n) {}
	int liczba;
};

Klasa operator+(Klasa &Kobj1, Klasa &Kobj2) { // referencja przyśpiesza działanie
	return Kobj1.liczba+Kobj2.liczba;
}

int main() {
	Klasa Kobj1(5), Kobj2(3);
	Klasa Kobj = Kobj1 + Kobj2;
}
```

Funkcja operatorowa będąca składową klasy wymaga by obiekt stojący po lewej stronie operatora był obiektem tej klasy.

```
Klasa Klasa::operator+(int n);
Klasa a;
a = a + 5; // a = 5 + a; ŹLE
```

Funkcja globalna nie ma tego ograniczenia.

```
Klasa operator+(int n, Klasa Kobj);
Klasa a;
a = 5 + a; // DOBRZE
```

Niektóre operatory muszą być funkcjami składowymi. Są to operator przypisania, nawiasy kwadratowe, nawiasy zwykłe oraz strzałka ->.

Operator przypisania = jest generowany automatycznie (co nie zawsze jest dobre), musi być składową klasy. Nie jest generowany automatycznie gdy klasa ma stały składnik, składnik będący referencją, ma składową klasę, w której operator przypisania jest prywatny, ma klasę podstawową z prywatnym operatorem przypisania. Nie jest dziedziczony.

Przeładowany operator przypisania, który nic nie robi:

```
Klasa& operator=(Klasa Kobj) { return *this; }
```

Operator [] – powinien mieć działanie podobne do tego, jakie ma dla typów wbudowanych. Można np. przypisać mu funkcję dostępu do konkretnego elementu np. w tablicy – bardzo intuicyjne.

```
#include <iostream>
class Klasa {
public:
	int operator[](int n) { return tablica[n]; }
private:
	int tablica[10]; // gdzieś w konstruktorze wypełniana liczbami
};

int main() {
	Klasa Kobj;
	std::cout << Kobj[5]; // wypisuje element o indeksie 5 z tablicy w obiekcie Kobj
}
```

Operator () – może posługiwać się dowolną liczbą parametrów, przydatny do robienia funktorów:

```
class Klasa {
public:
	int operator()(int a, int b) { return (a + b); }
};

int main() {
	Klasa Kobj;
	int n = Kobj(5,3); // Obiekt klasy Klasa zachowuje się jak funkcja - funktor
}
```

Operator -> jest bardzo rzadko używany, przydaje się przy pisaniu klasy udającej wskaźnik (wskaźnikor?).

Operator new i **delete** – przeciążone zawsze statyczne (nawet jeśli tego nie zadeklarujemy). Operator **new** musi zwracać typ void* i pobierać argument typu size_t. Delete przyjmuje void* i zwraca void.

```
#include <iostream>
class Klasa {
public:
	void* operator new(size_t rozm) {
		return new Klasa[rozm];
	}
	void operator delete(void* del) {
		delete del;
	}
};

int main() {
	Klasa *a = new Klasa;
	delete a;
}
```

Operatory inkrementacji i dekrementacji (zarówno pre jak i post) ++ -- działają jak zwykłe inne operatory jednoargumentowe.

```
class Klasa {
public:
	Klasa(int n) : liczba(n) {}
	Klasa& operator++() { ++liczba; return *this; } // preinkrementacja
	Klasa& operator++(int) { liczba++; return *this; } // postinkrementacja
	 int liczba;
};

int main() {
    Klasa Kobj(5); Kobj++; ++Kobj; }
```

Operatory << i >> najczęściej definiujemy w stosunku do klasy iostream, możemy je wtedy zdefiniować tylko globalnie, musi pracować na zmiennych globalnych, ewentualnie musi być zaprzyjaźniona z naszą klasą, jeżeli ma pracować na zmiennych prywatnych.

```
#include <iostream>
class Klasa {
public:
      Klasa(int n) : liczba(n) {}
      int liczba;
};

std::ostream& operator<<(std::ostream& strm, Klasa &Kobj) {
      return (strm << Kobj.liczba);
}

int main() {
      Klasa Kobj(5);
      std::cout << Kobj;
}
```

Przeładowanie operatora >> zostanie omówione w przyszłości.

Dziedziczenie – to technika umożliwiająca zdefiniowanie nowej klasy z wykorzystaniem klasy już istniejącej. Nowa klasa staje się automatycznie nowym typem danych. Klasa pochodna może mieć zdefiniowane dodatkowe dane składowe, nowe metody, redefiniować metody klasy bazowej.

```
class Bazowa {
public:
    int liczba;
};

class Pochodna : public Bazowa {
public:
    Pochodna(int n) { liczba = n; } // nie można użyć listy inicjalizacyjnej
};

int main() {
    Pochodna Pobj(5);
}
```

Do prywatnych składników klasy bazowej, klasa pochodna nie ma dostępu (chyba, że przy użyciu publicznych metod). Klasa pochodna ma również dostęp do elementów w zakresie **protected**, który to został wymyślony właśnie na potrzeby dziedziczenia.

```
class Pochodna : public Bazowa //
```
składniki public i protected niezmienione

```
class Pochodna : protected Bazowa //
```
składniki public i protected stają się protected

```
class Pochodna : private Bazowa //
```
odziedziczone składniki są teraz prywatną własnością klasy pochodnej

Deklaracja dostępu – jeśli klasa pochodna ukrywa wszystkie dziedziczone elementy (private), lecz mimo to chcemy zachować dostęp do pojedynczego elementu to należy o tym powiadomić używając operatora zakresu w sekcji public/protected.

```
class Bazowa {
public:
    int liczba;
    int liczba2;
};

class Pochodna : private Bazowa {
public:
    Bazowa::liczba; //  informujemy, że liczba jest dostępna publicznie
};

int main() {
    Pochodna Pobj;
    Pobj.liczba = 5;
    //Pobj.liczba2 = 5; //  błąd!  liczba2 jest dziedziczona jako prywatna
}
```

Elementy, które nigdy nie są dziedziczone to: konstruktor, operator przypisania, destruktor.

Dziedziczenie wielopoziomowe - ziedziczenie może zachodzić wielopoziomowo:

```
class Bazowa {
public:
    int liczba;
};

class Pochodna : public Bazowa {
public:
};

class DrugaPochodna : public Pochodna {
public:
    DrugaPochodna(int n) { liczba = n; }
};

int main() {
    DrugaPochodna DPobj(5);
}
```

Do pracy rusza najpierw konstruktor klas podstawowej, a dopiero potem konstruktor klasy pochodnej. Konstruktor klasy pochodnej powinien na liście inicjalizacyjnej wywoływać konstruktor klasy bazowej, chyba że taki nie istnieje albo klasa bazowa ma konstruktor domyślny.

```
Pochodna() : Bazowa(param) {}
```

Jeżeli klasa podstawowa posiada operator przypisania = to wygenerowany operator przypisania klasy pochodnej skorzysta z niego. Jeżeli klasa zawiera element const to taki operator nie jest generowany i trzeba go napisać.

Jeśli nie zdefiniowaliśmy konstruktora kopiującego to klasa pochodna wygeneruje go sobie sama. Dla obiektów const konstruktor zostanie wygenerowany tylko gdy klasy podstawowe i składniki klasy pochodnej gwarantują argumentowi nietykalność. W praktyce wszystkie konstruktory kopiujące powinny wyglądać tak:

```
Klasa(const Klasa &Kobj)
```

Omówienie wykładu 7: Funkcje wirtualne, dziedziczenie wielokrotne – czyli zabawy z dziedziczeniem ciąg dalszy.

Wskaźnik lub referencja do obiektu klasy pochodnej może być niejawnie przekształcony na wskaźnik lub referencję dostępnej jednoznacznie klasy podstawowej (o ile dziedziczenie jest publiczne).

```
class Bazowa {
public:
};

class Pochodna : public Bazowa {
public:
};

int main() {
      Pochodna Pobj;
      Bazowa &Bref = Pobj;
      Bazowa *Bptr = &Pobj;
}
```

Taka konwersja zachodzi np. przy przesyłaniu argumentów do funkcji, albo przy zwracaniu przez funkcję rezultatu jej działania. Mając zwrócić jedno, zwraca drugie. Analogicznie do sytuacji przy funkcjach, konwersja zachodzi również przy konstruktorach oraz w wyrażeniach inicjalizacyjnych.

```
class Bazowa {
public:
      Bazowa funkcja(Bazowa &Bobj) { return Bobj; }
};

class Pochodna : public Bazowa {
public:
};

int main() {
      Pochodna Pobj;
      Bazowa Bobj;
      Bobj = Bobj.funkcja(Pobj);
}
```

Uwaga! Do funkcji spodziewającej się adresu tablicy obiektów klasy podstawowej nie można wysłać adresu tablicy obiektów klasy pochodnej!

Funkcje wirtualne – przy nazwie funkcji pojawia się słowo virtual. Obiektówka pełną gębą. Możliwe jest ustawienia wskaźnika (referencji) typu klasy podstawowej tak, żeby pokazywał na obiekt klasy pochodnej. Wynika to z tego, iż obiekt klasy pochodnej jest szczególnym typem obiektu klasy bazowej.

```
class Bazowa {
public:
    virtual void funkcja() {}
};

class Pochodna : public Bazowa {
public:
    void funkcja() {}
};

int main()
{
    Pochodna Pobj;
    Bazowa *Bptr = &Pobj;
    Bptr->funkcja();  // zostanie wywołana funkcja z klasy Pochodna
}
```

Gdyby funkcja nie była wirtualna to została by wywołana wersja klasy Bazowej.
Wszystko to prowadzi do techniki zwanej **poliformizmem**. Jest to wykazywanie przez metodę różnych form działania w zależności od tego jaki typ obiektu aktualnie wskazywany jest przez wskaźnik lub referencję.

Klasa abstrakcyjna – to taka, która nie reprezentuje nic konkretnego. Jest stworzona tylko po to by po niej dziedziczyć. Tworzymy funkcje wirtualne, których będziemy używać, ale ich implementację zostawiamy tylko klasom pochodnym. Z klasami abstrakcyjnymi ścisły związek mają **funkcje czysto wirtualne**.

```
virtual void funkcja() = 0; // funkcja czysto wirtualna
```

Dopóki klasa ma choć jedną funkcję czysto wirtualną NIE MOŻNA stworzyć jakiegokolwiek jej obiektu.

```
class Bazowa {
public:
    virtual void funkcja() = 0;
};

class Pochodna : public Bazowa { public: void funkcja() {} };
class InnaPochodna : public Bazowa { public: void funkcja() {} };

int main() {
    Pochodna Pobj;
    InnaPochodna IPobj;
    Bazowa *Bptr = &Pobj;
    Bptr->funkcja(); // tu wywoła się wersja funkcji dla klasy Pochodna
    Bptr = &IPobj;
    Bptr->funkcja(); } // tu wywoła się wersja funkcji dla klasy InnaPochodna
```

Wirtualny destruktor – może istnieć mimo, iż jego nazwa w klasie pochodnej jest różna. Jest to jedyny wyjątek kiedy funkcje wirtualne mogą mieć różne nazwy. Klasa bazowa zawsze powinna definiować destruktor jako wirtualny.

dynamic_cast – operator rzutowania pozwalający przywrócić rzeczywisty typ obiektu. Jest to rzutowanie w dół (w przeciwieństwie do **static_cast** – rzutowania w górę).

```
class Bazowa {
public:
	virtual void funkcja() {}
};

class Pochodna : public Bazowa {
public:
};

int main()
{
	Pochodna Pobj;
	Bazowa *Bptr = static_cast<Bazowa*>(&Pobj);
	Pochodna *Pptr = dynamic_cast<Pochodna*>(Bptr);
	Pptr->funkcja();
}
```

typeid() – pozwala ustalić typ obiektu w czasie wykonywania programu

```
std::cout << typeid(obiekt).name();
```

Podobno powinno się dołączyć nagłówek typeinfo, ale bez niego też działa.

Dziedziczenie wielokrotne – klasa może dziedziczyć po więcej niż jednej klasie bazowej.

```
class Pochodna : Bazowa1, Bazowa2 { … };
```

Dzięki temu można połączyć ze sobą kilka niezależnych klas w jedną megaklasę.
Definicja klasy umieszczonej na liście klas bazowych musi być znana wcześniej, sama deklaracja wyprzedzająca nie wystarczy. Sposób dziedziczenia wskazuje się osobno dla każdej klasy bazowej.

```
class Pochodna : public Bazowa1, protected Bazowa2 { … };
```

.
Konstruktory wywoływane są zgodnie z listą klas bazowych.

Jeśli dwie klasy bazowe mają dokładnie ten sam element składowy to z klasy pochodnej odwołujemy się do niego przez operator zakresu. W wielokrotnym, wielopoziomowym dziedziczeniu widziana jest tylko składowa będąca najbliżej, ale zawsze się można pobawić zakresem. Nie będę tutaj wstawiał rysunków z Mamą i Dziadkiem z wykładu 7…

Dziedziczenie wirtualne – rozwiązuje problem gdy klasa ma dwie klasy bazowe, z których jedna dziedziczy z klasy bazowej klasy, której bazową sama jest (innymi słowy jeśli klasa A dziedziczy z B i C, a klasa B dziedziczy z C to klasa A miała by dostęp do klasy C dwukrotnie; jest to dopuszczalne, ale niedobre).

```
class Pochodna 1: virtual  public Bazowa { … };
class Pochodna2 : virtual  public Bazowa { … };
class DrugaPochodna : public Pochodna1, public Pochodna2 { … };
```

Ta sama klasa może być dziedziczona wirtualnie i niewirtualnie.

Klasa finalna – klasa, której nie można dziedziczyć.

```
class Bazowa {
private:
      ~Bazowa() {}
      friend class Pochodna;
};

class Pochodna :  virtual public Bazowa { };

class DrugaPochodna : public Bazowa { };

int main()
{
      Pochodna Pobj;
      //DrugaPochodna DPobj; //  nie przejdzie
}
```

Omówienie wykładu 8: Obsługa wyjątków – niewdzięczny temat, ale nie będzie wiele w jego zakresie. Zacznijmy od postaw: try, catch, throw. Najlepiej od razu przykład:

```
#include <iostream>
int main() {
      int liczba;
      try {
            throw liczba;
      }
      catch (int) {
            std::cout << "Uzycie throw przenosi nas tutaj.";
      }
}
```

Jeżeli w sekcji try napotkamy słowo throw to program znajdzie sekcję catch z podanym typem odpowiadającym typowi obiektu przez throw wyrzuconego. Try definiuje zakres, w którym wyjątki są przechwytywane. Throw przełącza tryb z normalnego do trybu obsługi wyjątków.

Bloków catch może być więcej. Wtedy każdy przystosowany jest do łapania wyjątków innego typu.

```
#include <iostream>
int main()
{
	int liczba;
	char litera;
	try {
		throw litera;
	}
	catch (int) { }
	catch (char)
	{
		std::cout << "Obsluga wyjatkow dla litery.";
	}
}
```

Throw w przeciwieństwie do zwykłych funkcji może wyrzucać (zwracać) obiekty dowolnego typu. Nie wraca też do programu w miejscu, w którym została wywołana (jak zwykła funkcja) tylko przenosi wykonywanie do bloku catch. Bloki catch są odnajdywane z góry na dół – nie ma znaczenia, która wersja jest lepiej dopasowana ze względu na typ danych. Jeśli istnieją dwie sekcje catch, które odpowiadają rzuconemu obiektowi to zawsze wybrana zostanie ta pierwsza.

Bloki try i catch można zagnieżdżać. Co więcej jeśli żadna z zagnieżdżonych sekcji catch nie odpowiada rzuconemu obiektowi to program poszuka takiej na zewnątrz (w sekcji catch będącej następstwem nadrzędnego zakresu try). Pokarzemy to na przykładzie:

```
#include <iostream>
int main() {
	int liczba;
	try {
		try {
			throw liczba;
		}
		catch (char) { }
	}
	catch (int) {
		std::cout << "Obsluga wyjatkow dla liczby”;
	}
}
```

Tutaj również obowiązuje zasada „kto pierwszy ten lepszy”.

Dana procedura obsługi nadaje się do pracy z rzuconym obiektem o określonym typie, jeżeli:
Typ obiektu rzuconego jest taki sam jak oczekiwany ; jeśli oczekiwany jest taki sam tylko, że const; gdy oczekiwany jest referencją do rzucanego; gdy oczekiwany jest publiczną klasą bazową w stosunku do rzucanego; gdy oba są wskaźnikami, a jedne może być skonwertowany do drugiego.

Pierwsze trzy przypadki nie wymagają komentarza. To po prostu działa. Pozostałe zaś obrazują następujące przykłady:

Oczekiwanie publicznej klasy bazowej:

```
#include <iostream>
class Bazowa { };
class Pochodna : public Bazowa { };

int main() {
	Pochodna Pobj;
	try {
		throw Pobj;
	}
	catch (Bazowa) {
		std::cout << "Obsluga wyjatkow dla klasy Pochodna";
	}
}
```

Odwikłanie stosu – oznacza, że po uruchomieniu procedury obsługi wyjątków program przywraca stan sprzed wejścia do sekcji try:

```
#include <iostream>
class Klasa {
public:
		~Klasa() { std::cout << "Dziala destruktor"; }
};

int main() {
	int liczba;
	try {
		Klasa Kobj;
		throw liczba; // tutaj zadziała destruktor obiektu typu Klasa ponieważ program
			      // przywróci stan sprzed wejścia do sekcji try, a więc zlikwiduje
			      // obiekty powstałe wewnątrz try – stos zostanie odwikłany
	}
	catch (int) { }
}
```

UWAGA: Obiekty stworzone za pomocą operatora new nie zostaną zlikwidowane (o ile wskaźnik do tego obiektu nie zostanie zlikwidowany w procesie przywracania stosu, tzn. został utworzony przed sekcją try)! By uporać się z likwidacją takiego obiektu można usunąć go manualne przed rzuceniem wyjątku lub zadbać by przeżył proces odwikłanie np. wyposażając rzucany obiekt w informacje o obiektach stworzonych operatorem new.

Obiekty automatyczne również zostają zlikwidowane, jednakże jeśli obiekt rzucany jest automatyczny to informacje w nim zawarte zostaną skopiowane podczas rzucania instrukcją throw.

Bardzo dobry przykład, pokazujący nieco zasad rządzących tym bałaganem:

```
#include <iostream>
class Klasa {
public:
	Klasa(int n) : l(n) { std::cout << "Konstruktor " << l << "\n"; }
	Klasa(const Klasa &Kobj) {
		l = Kobj.l;
		std::cout << "Konstruktor kopiujacy " << l << "\n";
	}
	~Klasa() { std::cout << "Destruktor  " << l << "\n"; }
	int l;
};

int main() {
	Klasa Kobj(1);			// 1
	try {
		Klasa Kobj2(2);		// 2
		throw Kobj;			// 3
	}
	catch (Klasa &Kref) {
		std::cout << "catch: " << Kref.l << std::endl; // 4
	}
}
```

Po kolei: **1** – działa zwyczajny konstruktor, **2** – tutaj również działa zwyczajny konstruktor, **3** – najpierw w ruch idzie konstruktor kopiujący (który kopiuje obiekt Kobj w celu nadania go do catch), a zaraz potem likwidowany zostaje obiekt utworzony wewnątrz try (Kobj2), **4** – kopia obiektu wysłanego do catch zostaje użyta wewnątrz tej sekcji, a następnie zlikwidowana. Obiekt Kobj jest likwidowany dopiero po zakończeniu programu. Polecam skompilować i przyjrzeć się działaniu.

Gdybyśmy instrukcji throw kazali rzucić obiekt Kobj2 (utworzony wewnątrz sekcji try) to i tak zostanie on zlikwidowany w tym samym miejscu.

Obiekty rzucane mogą być nieautomatyczne (globalne lub tworzone operatorem new), ale do sekcji catch i tak zawsze zostanie wysłana **kopia** (haha!).

Uwaga! Nie wolno rzucać wyjątków z destruktorów! Panuje zasada, iż kolejny wyjątek nie może być obsłużony, dopóki obsługiwany jest poprzedni.

```
#include <iostream>
class Klasa {
      public:
      ~Klasa() { throw liczba; }
      int liczba; };

int main() {
      int liczba;
      try {
            Klasa Kobj;
            throw liczba;
      }
      catch (int) { }
}
```

Program natychmiast **brutalnie zakończy działanie**. W procesie odwikłania stosu uruchomionym instrukcją throw, likwidowany zostaje obiekt Kobj (ponieważ jest utworzony wewnątrz try). W destruktorze tego obiektu rzucany jest inny wyjątek. Mamy dwa rzucone wyjątki. I co zrobić? Error.

Program powinien obsługiwać wszystkie rzucone wyjątki. Jeśli nastąpi sytuacja, w której nie da się takiego wyjątku obsłużyć uruchomiona zostaje funkcja **std::terminate()** i program kończy działanie. Ta sama funkcja działa w przypadku rzucania wyjątku w destruktorze, wystąpienia błędu w obsłudze wyjątków lub **konstruktor kopiujący między try i catch rzuci wyjątek** (taka sytuacja również jest niedopuszczalna).

Funkcję std::terminate() można zmodyfikować tak by nie kończyła działania programu (wywołując w swym ciele **std::abort()**). Używa się do tego funkcji **std::set_terminate**()

```
#include <iostream>
void funkcja() {
    std::cout << "Dzialanie funkcji\n";
    exit(-1);
}
int main() {
      void (*fptr)();
      fptr = funkcja;
      std::set_terminate(fptr);
      std::terminate();
}
```

Zmodyfikowana funkcja std::terminate() powinna mimo wszystko kończyć działanie programu (taki standard). Użycie takiej techniki jest przydatne np. podczas testowania działania programu.

Każda funkcja może deklarować co jest w stanie wyrzucić. Przydatne gdy mamy tylko deklaracje funkcji (np. z biblioteki) i chcemy wszystko zrobić elegancko:

```
void funkcja() throw(int, Klasa); // funkcja obiecuje rzucać tylko inty i obiekty Klasa
void funkcja() throw();           // funkcja obiecuje nie rzucać nic
```

Jeżeli mimo deklaracji, funkcja wyrzuci wyjątek, którego obiecała nie rzucać to wywołana zostanie funkcja **std::unexpected()**, która przez domniemanie prowadzi do std::terminate(). Jednak używając funkcji **std::set_unexpected()** możemy ustalić co ma zostać uczynione (mimo wszystko i tak program powinien zakończyć działanie lub ewentualnie rzucić nowy wyjątek – **std::bad_exception**).

```
#include <iostream>
void funkcja() throw(std::bad_exception) {
      throw 'a';
}
void wlasne_unexpected()
      std::cout << "Dziala funkcja unexpected\n";
}

int main() {
      void (*fptr)();
      fptr = wlasne_unexpected;
      std::set_unexpected(fptr);
      try {
            funkcja();
      }
      catch(int) { }
}
```

Taki program zakończy działanie, ale zmodyfikowana funkcja std::unexpected() wykona dodatkowe działania.

Klasy wyjątków – możemy sobie napisać własną klasę wyjątkową, która dziedziczy po klasie **exception** z biblioteki o tej samej nazwie. Obiekt takiej klasy zawiera dane na temat zaistniałego wyjątku. Generalnie są to zwykłe klasy, które używa się przy instrukcjach throw i catch. No to rozbudowany przykładzik:

```
#include <iostream>
#include <exception>
#include <string>
class MyException : public std::exception {
public:
      MyException(const std::string& what = "przyczyna") : wyjatek(what)
      {}
      const char* what() const throw() { return wyjatek.c_str(); }
      ~MyException() throw() {}
private:
    std::string wyjatek;
};

int main(void) {
    try {
        throw MyException("Wyjatek\n");
    }
    catch (std::exception &e)
    {
        std::cout << e.what();
    }
}
```

Omówienie programu: na początku definiujemy klasę MyException, która dziedziczy po exception z dołączonej biblioteki. Klasa ta wyposażona jest w konstruktor, który do zmiennej wyjatek zapisuje nasz opis wyjątku (lub przez domniemanie wpisuje tam „przyczyna"). Prócz tego nasza klasa wyposażona została w funkcję wypisującą treść opisu wyjątku. Reszta powinna być jasna.
UWAGA! Mój kompilator zażądał destruktora gwarantującego nie wyrzucanie wyjątków!

Omówienie wykładu 9: Szablony funkcji i klas

Szablon – pozwala na wielokrotne zdefiniowanie funkcji (albo klasy, jeśli jest to szablon klasy) wg, ściśle określonego schematu.

```
template <typename T> T suma(T a, T b) {
      return a + b;
}

int main() {
      int a = 3, b =5, c;
      c = suma(a, b);
}
```

Szablon funkcji realizującej dodawanie. Zmienna a i b mogą być dowolnego (ale tego samego) typu .

Szablon musi znaleźć się w zakresie globalnym. Jedyne różnice w generowanych funkcjach (lub klasach w przypadku szablonów klas) tkwią w typach argumentów.

Wiele parametrów szablonu – szablon może przyjmować zróżnicowane typy danych

```
template <typename T1, typename  T2> T1 suma(T1 a, T2 b) {
	return a + b;
}

int main() {
	int a = 3;
	float b = 5.4;
	int c;
	c = suma(a, b);
}
```

Szablon może również przyjmować zwykłe typy danych
.

```
template <typename T1, typename  T2> T2 funkcja(T1 a, T2 b, float c) {
	return (a + b)/c;
}

int main() {
	int a = 3;
	double b = 5.4;
	float c = 5;
	double d = funkcja(a, b, c);
}
```

Jeden szablon może być szczególnym przypadkiem drugiego:

```
template <typename T> T funkcja(T a, T b) { … }
template <typename T1, typname T2> T1 funkcja(T1 a, T2 b) { … }
```

To, który zostanie użyty zależy od sytuacji. Ewentualny konflikt nie prowadzi do błędu. Kompilator skorzysta (w przypadku gdy a i b są tego samego typu) z szablony dla jednego typu nieokreślonego.

W ciele szablonu możemy posługiwać się zarówno jego argumentem do tworzenia zmiennych automatycznych jak i **typów pochodnych** takich jak wskaźniki, referencje czy tablice:

```
template <typename T> T funkcja(T n) {
	T *ptr = &n;
	return *ptr;
}

int main() {
	int liczba = funkcja(5); }
```

Szablon może tworzyć funkcje z przydomkiem inline, static i extern.

```
template <typename T> inline T funkcja(T a) { … }
```

Globalna funkcja statyczna - taka funkcja ma zasięg lokalny tzn., że nie możemy używać jej w innej jednostce kompilacji.

Funkcje specjalizowane – funkcja generowana przez szablon może nie spełniać oczekiwań programisty. W takiej sytuacji definiujemy normalną funkcję, która dla określonych parametrów będzie dawała poprawny rezultat. Kompilator wykorzysta taką funkcję jeśli ją znajdzie. Jeśli nie – skorzysta z szablonu.

```
template <typename T> T funkcja(T a, T b) {
      return a + b ;
}

void funkcja(char a, char b) { } // funkcja specjalizowana

main() { }
```

Możemy tworzyć funkcję z szablonu i od razu deklarować jakiego typu ma ona być:

```
obiekt = funkcja<int>(a, b);
```

Szablony klas – podobnie jak funkcje, klasy również mogą powstawać z szablonów. Możemy dzięki temu otrzymać wiele podobnych klas tworząc jedynie schemat ich powstawania zależny od typów danych.

```
template <typename T> class Klasa {
public:
      T wartosc;
};

main() {
      Klasa<int> Kobj;
      Kobj.wartosc = 5;
}
```

Analogicznie do szablonów funkcji, parametrów szablonu klasy może być więcej

```
template <typename T1, typename T2> class Klasa { };
int main() {
      Klasa<int, char> Kobj;
}
```

Parametrem szablonu może być typ lub stałe wyrażenia, adresy

Funkcje składowe szablonu klasy – są definiowane wewnątrz szablonu klasy:

```
template <typename T> class Klasa {
public:
      void ustal_wartosc(T t) {
            wartosc = t;
      }
private:
      T wartosc;
};

main() {
      Klasa<int> Kobj;
      Kobj.ustal_wartosc(5);
}
```

Klasa szablonowa może być klasą bazową:

```
class Pochodna : public Bazowa<int> { … };
```

Szablon funkcji z argumentem będącym szablonem klasy - przypadek, w którym chcemy wygenerować funkcję pracującą na obiektach klasy powstałej z szablonu.

```
#include <iostream>
template <typename T> class Klasa {
public:
    void ustal_wartosc(T t) {
        wartosc = t;
    }
    T wartosc;
};

template <typename T> T ustal_i_wypisz_wartosc(T n) {
    Klasa<T> Kobj;
    Kobj.ustal_wartosc(n);
    std::cout << Kobj.wartosc;
}

main() {
    ustal_i_wypisz_wartosc(5);
}
```

Bardzo użyteczne jest stworzenie szablonu funkcji przeładowującej operator << w celu wypisania:

```
template <typename T>
std::ostream& operator<<(std::ostream &o, Klasa<T> &Kobj) {
	o << Kobj.wartosc << std::endl;
	return o;
}
```

Szablony klas nie mogą być zagnieżdżone. Nic nie stoi jednak na przeszkodzie by zdefiniować zwykłą klasę wewnątrz szablonu (nie odwrotnie, szablon nie może być w niczym zagnieżdżony).

Składnik statyczny działa tak jak w przypadku zwykłych klas, aczkolwiek jest odmienny dla każdego wygenerowanego szablonu. Trzeba go zdefiniować w przestrzeni globalnej (lub w przestrzeni nazw):

```
template <typename T> int K<typ>::skladnik_statyczny;
```

Klasa, tak jak funkcja, może być **specjalizowaną wersją szablonu.** W przypadku istnienia takiej specjalizowanej wersji klasy, kompilator najpierw spróbuje jej użyć, a dopiero w przypadku niepowodzenia użyje szablonu. Przy nazwie klasy powinno się pojawić template<>.

```
template<> class Klasa { … };
```

Klasa szablonowa może posiadać specjalizowaną funkcję składową.

Przyjaźń w szablonach klas – szablony klas tak jak zwykłe klasy mogą posiadać przyjaciół. Każda klasa może mieć swojego przyjaciela lub wszystkie klasy mogą mieć jednego - zależy to od tego jak zadeklarujemy przyjaźń. Jeśli przyjaciel zależy od parametru to każda klasa szablonowa ma swojego. To samo tyczy się funkcji. Logiczne.

Dziedziszenie klas szablonowych – klasa szablonowa może dziedziczyć.

```
template <typename T> class Bazowa {
public:
	T wartosc;
};

template <typename T> class Pochodna : public Bazowa<T> {
public:
};

class DrugaPochodna : public Pochodna<int> {
public:
};

main() {
	DrugaPochodna DPobj;
	DPobj.wartosc = 5; }
```

Omówienie wykładu 10: Operacje wejścia/wyjścia

Omówienie wykładu 11: Standardowa biblioteka szablonów (STL) - biblioteka dostarcza nam nowe możliwości zarządzania kolekcjami danych.

Kontenery – służą do zarządzania kolekcjami obiektów konkretnego typu.
Iteratory – służą do poruszania się po kolekcjach.
Algorytmy – służą do przetwarzania elementów kolekcji.

Bajki o dobrodziejstwach STLa zostawiam dla chętnych do samodzielnego poczytunku. Tutaj zajmiemy się zastosowaniem praktycznym.

Rodzaje kontenerów:
Kontenery sekwencyjne – reprezentują kolekcje uporządkowane, w których każdy element posiada określoną pozycje. Należą do nich: **Vector** - wektor, **Deque** – kolejka dwustronna, **List** - lista.
Kontenery asocjacyjne – reprezentują kolkcje sortowane, położenie elementu zależy od jego wartośći. Należądo nich: **Set**, **Multiset**, **Map**, **Multimap**.

Wspólne operacje dla wszystkich kontenerów:
ConType c – Pusty kontener
ConType c1(c2) – Inicjalizuje c2
ConType c1(beg,end) – Inicjalizuje zakresem
c.~ConType() – Zwalnia pamięć
c.size() – Liczba elementów
c.empty() – Czy element pusty?
c.max_size() – Maksymalna liczba elementów
==, !==, <, >, <=, >= – Operacje logiczne
c1 = c2 – Przypisanie
c1.swap(c2) – Zamiana elementów miejscami
swap(c1, c2) – Zamiana jako funkcja globalna
c.begin() – Iterator do pierwszego elementu
c.end() – Iterator do ostatniego elementu
c.rbegin() –
c.rend() –
c.insert(pos,ele) – Wstawia kopię elementu
c.erase(beg, end) – Usuwa element z podanego zakresu
c.clear() – Opróżnia kontener

Iteratory są obiektami, które potrafią nawigować po elementach kontenerów.
Podstaowe operacje:
operator* - Zwraca element z aktualnej pozycji
operator++ - Przesuwa iterator na następną pozycję
operator== i **operator!**= - Zwraca wartość logiczną czy iteratory reprezentują tę samą pozycję
operator= - Przypisanie

kontener::iterator – przeznaczony do nawigowania w trybie odczytu i zapisu
kontener::const_iterator – przeznaczony do nawigowania w trybie odczytu

Algorytmy – służą do przetwarzania elementów kolekcji czyli np. sortowania, kopiowanie, przestawiania. Są funkcjami globalnymi, a nie składowymi. Pracują na zakresach. W celu ich użycia inkludujemy bibliotekę **algorithm**.

Zakresy mogą stanowić cały kontener, ale nie muszą. Nie jest sprawdzana ich poprawność.

Wektor – każdy kto uczęszczał na laboratoria, na których można było z radością wpisać make nosend doskonale wie na jakiej zasadzie pracuje klasa **vector**. Dla tych, którzy nie mieli tej przyjemności wyjaśniam, iż wektor to coś w rodzaju tablicy pozwalającej na swobodny dostęp za pomocą indeksu, szybkie dodawanie i usuwanie elementów (początkowego i końcowego, inne już nie takie szybkie).

W celu wykorzystania wektora należy dołączyć do programu bibliotekę **<vector>**.
Operacje:
vector<typ> v – Tworzy pusty wektor składający się z elementów podanego typu
vector<typ> v1(v2) – Tworz wektor będący kopią wektora podanego w nawiasie
vector<typ> v(liczba) – Tworzy wektor o podanej ilości elementów
vector<typ> v(liczba,wartość) – Tworzy wektor o podanej ilości elementów które przyjmują wartość
vector<typ> v(beg,end) – Tworzy wektor inicjalizowany elementami z podanego zakresu. Przykład:

```
#include <iostream>
#include <vector>
int main() {
	vector<int> vec(3);
	vec[0] = 1; vec[1] = 2; vec[2] = 3;

	vector<int> vec2(vec.begin(), vec.end());

	for(int i = 0; i<3; i++) {
		std::cout << vec2[i] << std::endl;
	}
}
```

Operacje niemodyfikujące: **v.empty()**, **v.size()**, **v.max_size()** – wspólne z innymi kontenerami; **v.capacity()** – zwraca pojemność wektora bez realokacji; **v.reserve(n)** – rezerwuje pamięć dla **n** elementów.

Operacje przypisania:
v1 = v2 – Chyba nie wymaga komentarza
v.assign(n, wartość) – n elementom wektora przypisuje podaną wartość
v.assign(beg,end) – Przypisuje element z podanego zakresu
v1.swap(v2), swap(v1, v2) – Zamienia v1 z v2

Operacje dostępu:
v.at[indeks] – Zwraca element o podanym indeksie lub zwraca wyjątek jeśli nie zgadza się zakres
v[indeks] –Zwraca element o podanym indeksie bez kontroli zakresu
v.front() – Zwraca pierwszy element
v.back() – Zwraca ostatni element

Funkcje iteratorowe:
v.begin() – Zwraca iterator dostępu swobodnego dla pierwszego elementu
v.end() – Zwraca iterator dostępu swobodnego dla pozycji za ostatnim elementem
v.rbegin() – Zwraca iterator odwrotny dla pierwszego elementu iteracji odwrotnej
v.rend() – Zwraca iterator odwrotny za ostatnim elementem iteracji odwrotnej
Operowanie na wektorach może odbywać się również tak jak na zwykłych tablicach.

Operacje wstawiania i usuwania elementów:
v.insert(pos, elem) – Wstawia podany element na podaną pozycję, zwraca pozycję nowego elementu
v.insert(pos, n, elem) – Wstawia podaną liczbę podanych elementów na podaną pozycję
v.insert(pos, beg, end) – Wstawia na podaną pozycję kopię elementów z podanego zakresu
v.push_back(elem) – Wstawia kopię podanego elementu na koniec wektora
v.pop_back() – Usuwa ostatni element
v.erase(poz) – Usuwa element na podanej pozycji i zwraca pozycję następnego elementu
v.erase(beg, end) – Usuwa element z podanego zakresu i zwraca pozycję następnego
v.resize(n) – Zmienia rozmiar wektora na podany
v.resize(n, elem) – Zmienia rozmiar wektora na podany i wypełnia go elementami o podanej wartości
v.clear() – Opróżnia wektor

Pojemność wektora nigdy nie maleje (nawet przy usuwaniu jego elementów). Zajmuje ciągły obszar w pamięci. Zapewniają minimalną obsługę wyjątków (jedynie funkcja at()).

Poniżej znajdziecie przykład prezentujący kilka podstawowych operacji związanych z wektorami:

```
#include <iostream>
#include <vector>
void dane_wektora(std::vector<int> &vec);

int main () {
      std::vector<int> vec(10, 5);
      dane_wektora(vec);
      vec.clear();
      std::vector<int> vec2(10, 123);
      vec = vec2;
      dane_wektora(vec);
      std::cout << vec.front() << ", " << vec.back() << std::endl;
}

void dane_wektora(std::vector<int> &vec) {
    std::cout << vec.size() << ", " << vec.capacity() << std::endl;
}
```

Kolejka dwustronna – bardzo podobna do wektora. Najważniejsze różnice to: kolejka może zmaleć, dodawanie elementów na początku i końcu jest szybkie, operacje dostępu i ruchu iteratora są nieco wolniejsze. Kolejka nie udostępnia funkcji capacity() i reserve()!

Możliwości kolejek dwustronnych w stosunku do wektorów:
d.push_front(elem) – Umieszcza podany element na początku kolejki
d.pop_front() – Usuwa pierwszy element kolejki

Lista – jej elementy są zorganizowane w postaci listy dwukierunkowej (Podstawy programowania – ah, aż się łezka w oku kręci). Można poruszać się wzdłuż jej elementów, iteratory są dwukierunkowe. W celu wykorzystania listy należy dołączyć bibliotekę **list.**

Lista nie zapewnia dostępu swobodnego. By dostać się do konkretnego elementu musimy przejść po wszystkich wcześniejszych. Wstawianie i usuwanie elementów ze środka listy jest szybkie. Usuwanie i operowanie na elementach nie unieważnia wskaźników i referencji do nich, ani do innych.

Operacje tworzenia i przypisania dla listy są takie same jak dla wektora i kolejki dwustronnej. Operacje modyfikujące są takie same jak dla kolejki (nie ma reserve() i capacity()).

Dodatkowe informacje:
l.front() – Zwraca pierwszy element listy
l.back() – Zwraca ostatni element listy

l.remove(wartosc) – Usuwa element o podanej wartości
l.remove_if(op) – Usuwa elementy, dla których wywołanie **op(element) == true**

l.splice(pos, l2) – Przenosi wszystkie elementy **l2** do **l** i wstawia je przed podaną pozycją
l.splice(pos, l2, l2pos) – Przenosi elementy z podanej pozycji z **l2** do **l** i umieszcza je przed **pos**
l.splice(pos, l2, l2beg, l2end) – Przenosi element z podanego zakresu w stosowne miejsce
l.merge(l2) – Przy założeniu, że obie listy są posortowane, przenosi elementy z **l2** do **l**
l.merge(l2, op) – Tak jak wyżej tylko pod warunkiem, że **op() == true**

Inne funkcje:
l.sort() – Sortuje listę
l.sort(op) – Sortuje listę używając **op()** od porównania
l.unique() – Usuwa powtórzenia kolejnych elementów o tej samej wartości
l.unique(op) – Tak jak wyżej tylko pod warunkiem, że **op() == true**
l.reverse() – Odwraca kolejność wszystkich elementów listy

Późno już. Jak mi się zachce to walnę tutaj jakiś przykład.

Omówienie wykładu 12: Funktory, zbiory, mapy, iteratory

Predykaty – jest to specjalny rodzaj funkcji (obiektu funkcyjnego – funktora). Zawsze zwracają wartość true albo false. Służą do określania kryteriów np. sortowania, wyszukiwania, itp.
Mogą być jedno lub dwuargumentowe. Dla tego samego wywołania zwracają zawsze tę samą wartość.

```
#include <iostream>
#include <list>

bool to_piec (int n) {
	if (n == 5) return true;
}

int main() {
	std::list<int> coll;
	for (int i = 0; i < 10; i++) coll.push_back(i);
	std::list<int>::iterator pos;
	pos = find_if (coll.begin(), coll.end(), to_piec);
	std::cout << *pos;
}
```

Powyższy przykład tworzy listę liczb i wstawia do niej dziesięć elementów. Następnie po zadeklarowaniu iteratora służącego do poruszania się po naszej liście, wpisuje do niego efekt działania funkcji **find_if**, która znajduje (używając predykatu jednoargumentowego) pozycję spełniającą warunek.

Funktory – inaczej obiekty funkcyjne. Są to obiekty, których działanie ma przypominać działanie funkcji. Jeden funktor już został zaprezentowany wcześniej (klasa posiadająca przeładowany operator nawiasów). Zaletą obiektów takich klas jest możliwość przechowywania dodatkowych danych. Dodatkowo obiekt funkcyjny posiada własny stan zanim zostanie wykonany, więc jego wywołanie nawet z tym samym argumentem (lub argumentami) może dawać różny rezultat.

Predefiniowane obiekty funkcyjne w nagłówku **functional**:
negate<T>() - - parametr
plus<T>() – parametr + parametr
minus<T>() – parametr - parametr
multiplies<T>() – parametr * parametr
divides<T>() – parametr / parametr
modules<T>() – parametr % parametr
equal_to<T>() – parametr == parametr
not_equal_to<T>() – parametr != parametr
less<T>(), less_equal<T>() – parametr < parametr i parametr <= parametr
greater<T>(), grater_equal<T>() – parametr > parametr i parametr >= parametr
logical_not<T>(), logical_and<T>(), logical_or<T>() - ! && ||

```
#include <iostream>
#include <functional>

int main() {
	std::logical_not<bool> odwroc_stan;
	bool stan = false;
	if(odwroc_stan(stan)) std::cout << "Wiadomosc" << std::endl;
}
```

Pary – obiekty klasy **pair** umożliwiające potraktowanie dwóch wartości jako pojedynczego elementu. Klasa ta jest zdefiniowana w nagłówku **utility**.

```
#include <iostream>
#include <utility>

int main() {
		std::pair<int, int> para_liczb;
		para_liczb.first = 1; para_liczb.second = 2;
		std::cout << para_liczb.first + para_liczb.second << std::endl;
}
```

Zbiory i wielozbiory – kontenery **set** i **multiset**. Wykonują automatyczne sortowanie swoich elementów. Różnica polega na tym, iż wielozbiory dopuszczają powtarzanie się elementów, a zbiory nie. Wymagają dołączenia biblioteki **set**. Domyślne kryterium sortowania to operator mniejszości reprezentowany przez obiekt funkcyjny less.

Najczęściej służą do reprezentowania drzew binarnych. Największą zaletą jest szybkie wyszukiwanie, ale z drugiej strony by zmienić element, trzeba usunąć go i dopiero wstawić nowy.
Zbiory nie udostępniają operacji bezpośredniego dostępu do elementów. Więcej w wykładzie 12.

Operacje tworzenia:
set<T> s – Tworzy pusty zbiór
set<T> s1(s2) – Tworzy kopię zbioru
set<T> s(op) – Tworzy zbiór o kryterium **op**
set<T> s(beg, end) – Tworzy zbiór i inicjalizuje go podanym zakresem
set<T> s(beg, end, op) – Tworzy zbiór i inicjalizuje go podanym zakresem o kryterium sortowania **op**

Listę można deklarować używając:
set<T> - Sortowanie używając **less**
set<T, op> - Sortowanie używając **op**
multiset<T> - Sortowanie używając **less**
multiset<T, op> - Sortowanie używając **op**
Użycie kryterium sortowania jako parametru konstruktora świadcz o możliwości jego późniejszej zmiany.

Operacje przypisania są zupełnie standardowe. Istnieje możliwość użycia funkcji swap().

Iteratory:
s.begin() – Zwraca iterator dwukierunkowy dla pierwszego elementu
s.end() – Zwraca iterator dwukierunkowy dla ostatniego elementu
s.rbegin() – Zwraca iterator odwrotny dla pierwszego elementu iteracji odwrotnej
s.rend() – Zwraca iterator odwrotny dla pozycji za ostatnim elementem iteracji odwrotnej

Wstawianie i usuwanie:
s.insert(elem) – Wstawia kopię elementu, zwraca stan wykonania operacji
s.insert(pos, elem) – Wstawia kopię, zwraca pozycję nowego, a podana służy jako wskazówka
s.insert(beg, end) – Wstawia kopię elementów z podanego zakresu
s.erase(wart) – Usuwa wszystkie elementy o podanej wartości, zwraca ile ich było
s.erase(pos) – Usuwa element o podanej pozycji
s.erase(beg, end) – Usuwa podany zakres
s.clear() – Opróżnia zbiór

```
#include <iostream>
#include <set>
#include <functional>

template <typename T> class wiekszy {
public:
      bool operator()(const T& a, const T& b) {
            if(a > b) {
                  return true;
            }
            else {
                  return false;
            }
    }
};

int main () {
      std::set<int,std::wiekszy<int> > s;
      std::set<int>::iterator pos;
      for (int i = 0; i < 10; i ++) s.insert(i);
      s.insert(5);
      for(pos = s.begin(); pos != s.end(); ++pos) {
            std::cout << *pos << std::endl;
      }
}
```

W powyższym programie definiujemy sobie zbiór liczb, a jego kryterium sortowania jest funkcjonał napisany na wzór grater. Dzięki temu liczby w zbiorze będą przechowywane od największej do najmniejszej. Definiujemy sobie iterator do poruszania się po zbiorze, a następnie wstawiamy do zbioru dziesięć wartości. Próba wstawienia wartości, która już tam się znajduje nie spowoduje jakichkolwiek działań. Ostatecznie wypisujemy zawartość zbioru.

Mapy i multimapy – są kontenerami, w których elementy przechowywane są parami klucz-wartość. W multimapach dopuszczalne by klucze się powtarzały. Oba typy kontenerów wymagają dołączenia biblioteki **map**. Domyślny iterator to – tak jak w zbiorach – less.

map<typ_klucza, typ_wartosci> m – Opcjonalnie trzeci argument określa kryterium sortowania.

Para klucz-wartość musi umożliwiać operację przypisania oraz kopiowania. Klucz musi być porównywalny za pomocą kryterium sortowania. Interfejs bardzo podobny do zbiorów i wielozbiorów

Operacje tworzenia:
map<K, W> m – Tworzy pustą mapę
map<K, W> m(m2) – Tworzy kopię podanej mapy
map<K, W> m(op) – Tworzy mapę o kryterium sortowania **op**
map<K, W> m(beg, end) – Tworzy mapę inicjalizowaną elementami z podanego zakresu
map<K, W> m(beg, end, op) – Tak jak powyżej + **op** jest kryterium sortowania

Operacje niemodyfikujące, przypisania, funkcje iteratorowe, wstawiające i usuwające
dla map i multimap są dokładnie takie same jak dla zbiorów i multizbiorów

Operacje wyszukiwania:
m.count(klucz)- Zwraca liczbę elementów o podanym kluczu
m.find(klucz) – Zwraca pozycję pierwszego elementu o podany kluczu lub **m.end()**
m.lower_bound(klucz) – Zwraca pierwszą pozycję, której klucz jest większy bądź równy podanemu
m.upper_bound(klucz) – Zwraca pierwszą pozycję, której klucz jest większy niż podany
m.equal_range(klucz) - Zwraca parę, która jest wynikiem działania dwóch powyższych funkcji
m.value_comp() – Zwraca obiekt, służący jako kryterium porównania wartości
m.key_komp() – Zwraca kryterium porównania wartości

Używając funkcji składowej insert należy pamiętać, iż wstawiamy parę.
m.insert(pair<string, int>(„Opis”, wartość))
m.insert(make_pair(„Opis”, wartość))
m.insert(map<string, int>::value_type(„Opis”, wartość))

Zastosowanie map i multimap: Można ich użyć jako tablic asocjacyjnych. Są to takie tablice, w której kluczem może być dowolny typ, nie tylko liczba naturalna. Indeks w takim przypadku zawsze musi być prawidłowy.

```
#include <iostream>
#include <map>
int main () {
      std::map <int, std::string, std::greater<int> > m;
      m.insert(std::pair<int, std::string>(3, "Trzy"));
      m.insert(std::make_pair(1, "Jeden"));
      m.insert(std::map<int, std::string>::value_type(2, "Dwa"));
      std::map<int, std::string>::iterator pos;
      for(pos = m.begin(); pos != m.end(); pos++)
            std::cout << pos->second << std::endl;
}
```

Iterator – raz jeszcze omówimy temat iteratorów, ale skupimy się na poznawaniu nowych rzeczy względem już zdobytych informacji.
Istnieje pewna grupa specjalnych definicji iteratorów jak np. iterator odwrotne. Do ich użycia konieczne jest dołączenie biblioteki **iterator.**

Funkcje pomocnicze dla iteratorów - są trzy:
void advance(pos, n) – Przesuwa pozycję iteratora o podaną liczbę
long distance(pos1, pos2) – Oblicza odległość między itereatorami
void iter_swap(pos1, pos2) – Zamienia wartości, do których odnoszą się podane iteratory

Adaptatory iteratorów – są to specjalne iteratory umożliwiające działanie algorytmom w specjalnym trybie. Należą do nich iteratory odwrotne, wstawiające, strumieniowe.

Iterator wstawiający – pozwalają algorytmom na zmianę operacji przypisania w operacji wstawiania
Przykład:

```
#include <iostream>
#include <list>
#include <iterator>

int main () {
      std::list<int> l(3);

      std::back_insert_iterator<std::list<int> > back_iter(l);
      std::front_insert_iterator<std::list<int> > front_iter(l);
      back_iter = 1;
      front_iter = 1;

      std::list<int>::iterator pos = l.end();
      std::insert_iterator<std::list<int> > insert_iter(l, pos);
      insert_iter = 2;

      pos = l.begin();
      insert_iter = std::insert_iterator<std::list<int> >(l, pos);
      insert_iter = 2;

      for(pos=l.begin(); pos!=l.end(); pos++) std::cout<< *pos <<std::endl;
}
```

Polecam kompilację, a sam zabieram się za wyjaśnienie: Na początku deklarujemy nowiutką listę trzyelementową. Następnie tworzymy dwa iteratory wstawiające **back_insert_iterator** i **front_insert_iterator.** Pierwszy zawsze wstawia element na koniec, a drugi na początek listy, do z którą jest związany. Elementy wstawia do listy na drodze prostej operacji przypisania. Kolejny iterator wstawiający – **insert_iterator** – działa na tej samej zasadzie, z tą różnicą, iż należy określić pozycję, do której będzie wstawiał nowy element. Zaznaczyć należy, że powyższe iteratory dodają elementy do listy, a nie podmieniają istniejące.

UWAGA! Możliwość użycia iteratora **front_iterator** jest uzależnione od tego czy możemy użyć funkcji push_front**.** Funkcja ta nie jest realizowana np. dla wektorów.

Iterator strumieniowy – umożliwia wykorzystanie algorytmów do pracy ze strumieniami.

```
#include <iostream>
#include <vector>
#include <iterator>

int main () {
	std::vector<int> v(5, 3);
	std::ostream_iterator<int> oi(std::cout, ", ");
	oi = 1;
	oi = 2;
	oi = 3;
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

Tworzymy wektor o pięciu elementach, które inicjalizujemy wartością 3. Następnie tworzymy iterator strumieniowy, który wiążemy ze strumieniem wyjściowym i wyposażamy w separator. Od tej pory używając przypisania z utworzonym operatorem wypisujemy na ekran podane wartości. Warto pokazać też przykład zastosowania w funkcji **copy()** (która jeszcze zostanie opisana) by ułatwić sobie wypisywanie na ekran zawartości całego wektora.

```
#include <iostream>
#include <vector>
#include <iterator>

int main () {
	std::vector<int> v;
	std::back_insert_iterator<std::vector<int> > back_insr(v);
	std::cout << "Wpisuj kolejne liczby. Wpisanie 0 konczy program\n";
	std::istream_iterator<int> liczba(std::cin);
	while(*liczba) {
		back_insr = *liczba;
		advance(liczba, 1);
	}
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

Powyższy program pozwala wypełnić wektor liczbami z klawiatury, podawanymi tak długi, aż użytkownik wpisze zero. Na końcu wypisuje wszystko na ekran.

Iterator odwrotny – działa tak jak zwykły tylko traktuje elementy w odwrotnej kolejności
kontener<T>::reverse_iterator
kontener<T>::const_reverse_iterator
rbegin() jest odpowiednikiem **begin()**, a **rend()** odpowiednikiem **end()**

Możliwa jest konwersja iteratorów na iteratory odwrotne:
kontener<T>::iterator pos;
kontener<T>::reverse_iterator rpos(pos);
Konwersja w drugą stronę możliwa jest przy użyciu funkcji **base()**

Znany nam już przykład, tym razem z iteratorem odwrotnym:

```
#include <iostream>
#include <map>
#include <iterator>
int main () {
	std::map <int, std::string, std::greater<int> > m;
	m.insert(std::pair<int, std::string>(3, "Trzy"));
	m.insert(std::make_pair(1, "Jeden"));
	m.insert(std::map<int, std::string>::value_type(2, "Dwa"));
	std::map<int, std::string>::reverse_iterator rpos;
	for(rpos = m.rbegin(); rpos != m.rend(); rpos++)
		std::cout << rpos->second << std::endl;
}
```

Omówienie wykładu 13: Algorytmy STL – tutaj omówię jedynie funkcje, które obowiązkowo trzeba znać na egzamin. Należą do nich: **for_each(), copy(), find(), remove(), unique(), sort(), accumulate(), binary_search(), remove_if()**

Algorytmy są zapisane w znanej nam już bibliotece algorithm.

for_each(beg, end, op) – Dla podanego zakresu wywołuje podaną operację

```
#include <iostream>
#include <deque>
#include <iterator>

class A {
public:
	A(): i(1) {};
	void operator()(int &el){
		el += i++;
	}
	int i;
};

int main() {
	std::deque<int> l(3);
	std::for_each(l.begin(), l.end(), A());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

copy(srcBeg, srcEnd, destBeg) – Kopiuje elementy z podanego przedziału na pozycję zaczynającą się od **destBeg**.

```
#include <iostream>
#include <deque>
#include <iterator>

int main() {
	std::deque<int> l(10,0);
	std::deque<int> l2(10,1);
	std::copy(l2.begin()+5, l2.end(), l.begin()+1);
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

find(beg, end, war) – Zwraca pozycję pierwszego elementu równego podanej wartości

```
#include <iostream>
#include <deque>

int main() {
	std::deque<int> l;
	for(int i=1; i<11; i++) l.push_back(i);
	std::deque<int>::iterator pos;
	pos = find(l.begin(), l.end(), 5);
	std::cout << *pos;
}
```

find_if(beg, end, op) – Zwraca pozycję pierwszego elementu, dla którego spełniony jest warunek.

```
#include <iostream>
#include <list>

bool to_piec (int n) {
	if (n == 5) return true;
}

int main() {
	std::list<int> coll;
	for (int i = 0; i < 10; i++) coll.push_back(i);
	std::list<int>::iterator pos;
	pos = find_if (coll.begin(), coll.end(), to_piec);
	std::cout << *pos;
}
```

remove(beg, end, war) - Usuwa pierwszy element o podanej wartości

```
#include <iostream>
#include <deque>
#include <iterator>

int main() {
	std::deque<int> l;
	for(int i=1; i<11; i++) l.push_back(i);
	remove(l.begin(), l.end(), 5);
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

remove_if(beg, end, op) – Usuwa pierwszy element, który spełnia podany warunek

```
#include <iostream>
#include <deque>
#include <iterator>

bool to_piec (int n) {
	if (n == 5){return true;}
	else { return false; }
}

int main() {
	std::deque<int> l;
	for(int i=1; i<11; i++) l.push_back(i);
	remove_if(l.begin(), l.end(), to_piec);
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

unique(beg, end) – Usuwa powtórzenia **sąsiadujące**

```
#include <iostream>
#include <deque>
#include <iterator>

int main() {
	std::deque<int> l;
	for(int i=1; i<10; i++) l.push_back(1);
	std::deque<int>::iterator pos;
	pos = std::unique(l.begin(), l.end());
	l.resize(pos - l.begin());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

unique(beg, end, op) – Usuwa wszystkie powtórzenia **sąsiadujące**, dla których **op(elem, e) == true** (wszystkie elementy po **e**, dla których **op** zwraca prawdę.

```
#include <iostream>
#include <deque>
#include <iterator>

bool porownanie(int a, int b) {
    return a==b;
}

int main() {
      std::deque<int> l;
      for(int i=1; i<10; i++) l.push_back(1);
      std::deque<int>::iterator pos;
      pos = std::unique(l.begin(), l.end(), porownanie);
      l.resize(pos - l.begin());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

sort(beg, end) – Sortuje podany zakres (domyślnie od najmniejszego do największego)

```
#include <iostream>
#include <deque>
#include <iterator>

int main() {
      std::deque<int> l;
      l.push_back(2); l.push_back(3); l.push_back(1);
      std::sort(l.begin(), l.end());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

sort(beg, end, op) – Sortuje podany zakres (biorąc jako kryterium dwuargumentowy predykat).

```
#include <iostream>
#include <deque>
#include <iterator>
#include <functional>

int main() {
      std::deque<int> l;
      l.push_back(2); l.push_back(3); l.push_back(1);
      std::sort(l.begin(), l.end(), std::greater<int>());
std::copy(l.begin(), l.end(), std::ostream_iterator<int>(std::cout, ", "));
}
```

accumulate(beg, end, war) – Zwraca sumę wszystkich elementów z podanego zakresu, dodając do niej wartość **war**. Wymaga dołączenia nagłówka **numeric**.

```
#include <iostream>
#include <deque>
#include <numeric>

int main() {
      std::deque<int> l;
      for(int i=0; i<3; i++) l.push_back(i);
      std::cout << std::accumulate(l.begin(), l.end(), 5);
}
```

accumulate(beg, end, war, op) – Zwraca wynik podanej operacji dla wszystkich elementów z podanego zakresu. Najłatwiej zrozumieć analizując przykład:

```
#include <iostream>
#include <deque>
#include <numeric>
#include <functional>

int main() {
     std::deque<int> l;
     for(int i=0; i<3; i++) l.push_back(i);
     std::cout << std::accumulate(l.begin(), l.end(), 5, std::plus<int>());
}
```

binary_search(beg, end, war) - Zwraca wynik poszukiwania podanej wartości w podanym zakresie

```
#include <iostream>
#include <deque>

int main() {
      std::deque<int> l;
      for(int i=0; i<10; i++) l.push_back(i);
      if(std::binary_search(l.begin(), l.end(), 3)) std::cout << "OK";
}
```