Dynamiczna alokacja pamięci

Tworzenie zmiennej na stercie - operator new.

```
int * p = new int;
```

Zmienna nie znika, można z niej korzystać gdzieś indziej. Nie ma ona nazwy, zawsze trzeba do niej dotrzeć przez wskaźnik.

Uważać na wyciek pamięci - nieusunięta zmienna ze sterty. Należy używać biblioteki do kontroli pamięci, aby widzieć wycieki.

Każda zmienna ze stosu musi być zwolniona operatorem delete, która przyjmuje za zmienną wskaźnik.

Tablice są zwalnianie w inny sposób:

```
delete [ ] tab;
```

Przypisanie adresu wskaźnika do danej komórki pamięci:

```
int * w = (int *) 123;
```

Zmienne na stercie są potrzebne do przechowywania danych nawet po skończeniu funkcji.

Lista jednokierunkowa

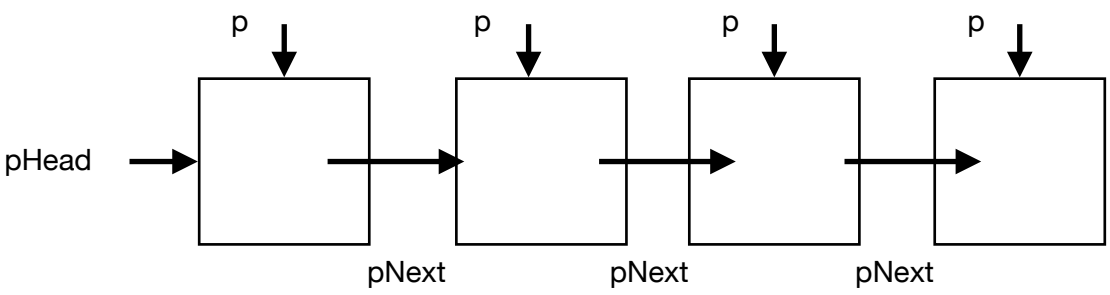
Jeśli lista jest pusta, wskaźnik na listę to null.

Wówczas tworzymy nowy element i modyfikujemy wskaźnik.

```
if (pHead = nullptr) // jeśli lista jest pusta
{
        element * pNowy = new element;
        (*pNowy).wartosc = liczba;
        pNowy -> pNext = nullptr;


}
else // jeśli coś w niej jest
{
        element pNowy * = new element;
        pNowy -> wartosc = liczba;
        pNowy -> pNext = pHead;
        pHead = pNowy;
}Listę można wypisać iteracyjnie lub rekurencyjnie.
```

Zawsze sprawdzać pustość listy.



Jeśli istnieje następny element, wskaż na niego. p się przesuwa

Fikuśne wypisywanie:

- sprawdzamy, czy w liście coś jest, a nie czy nie ma (if (pHead == nullptr) => if (pHead)
- dzięki operowaniu na kopii usuwamy p i działamy na pHeadzie - zmiany nie zostaną zachowane po wykonaniu funkcji
- trzeba jednak sprawdzać, czy lista nadaje się do użycia w tej wersji - wypisać dwa razy listę
- pozbyć się wycieków pamięci

```
if (pHead)
{
	while (pHead)
	{
		std::cout << pHead->wartosc << „ „;
		pHead = pHead -> pNext;
	}
}
```

Usunięcie wycieków:

- nie wolno usuwać elementu, a następnie się przesunąć, trzeba zapamiętać adres następnego elementu przed usunięciem
- ponownie robić test - 2 razy wypisywać funkcję dla pewności

```
void usunListeIteracyjnie (element * & pHead)
{

	while (pHead)
	{
		element * pNastepnik = pHead -> pNext;
		delete pHead;
		pHead = pNastepnik;
	}

}
```

Dodawanie elementów - tym razem na końcu listy.

```
void dodajNaKoniecIteracyjnie (element * & pHead, typ liczba)
{
	if (not pHead) // lista pusta
	{
		pHead = new element { liczba, nullptr };
	}
	else // lista cos zawiera
	{
		element * pNowy = new element { liczba, nullptr };
		auto p = pHead;
		while ( p->pNext )
		{
			p = p -> pNext;
		}
		p -> pNext = pNowy;
	}
}
```

```
void dodajNaKoniecRekurencyjnie ( element * & pHead, typ liczba)
{
	if (not pHead)
	{
		pHead = new element { liczba, nullptr };
	}
	else
	{
		if ( pHead -> pNext == nullptr ) // ostatnielement w liscie, dodajemy za nim
			pHead -> pNext = new element { liczba, nullptr};
		else // jestesmy w srodku
			{
				dodajNaKoniecRekurencyjnie (pHead -> pNext, liczba);
			{
	}
}
```

pNext wewnątrz funkcji traktowane jest jako pHead.

Wersja fikuśna - uwaga na egzamin! Zadanie typu „zrobić coś na liście bez pętli”.

```
void dodajNaKoniecRekurencyjnie ( element * & pHead, typ liczba)
{
	if (not pHead)
		pHead = new element { liczba, nullptr };
	else
		dodajNaKoniecRekurencyjnie (pHead -> pNext, liczba };
}
```

Ta wersja grozi nadmiernym wykorzystaniem pamięci - stack overflow.

Wypisanie listy od końca:

```
void wypiszOdKoncaRekurencyjnie ( element * pHead, std:: ostream & ss)
{
	if (pHead)
	{
		wypiszOdKoncaRekurencyjnie (pHead -> pNext, ss); // wypisz ogon
		ss << pHead -> wartosc << „ „; // wypisz mnie - pierwszy
	}
}
```