Stos liczb całkowitych

```
class StosInt {
  int* tab;
  unsigned size ;
  unsigned top ;
public:
  StosInt(unsigned s=10)
    {tab=new int[size=s];top=0;}
  ~StosInt(){delete[] tab;}

  void push(int i){tab[top++]=i;}
  int  pop(void) {return tab[--top];}
  int  empty() {return !top;}
};
```

Stos napisów

```
class StosString {
  string* tab;
  unsigned size ;
  unsigned top ;
public:
  StosString(unsigned s=10)
    {tab=new string[size=s];top=0;}
  ~StosString(){delete[] tab;}
  void push(string el){tab[top++]=el;}
  string  pop(void) {return tab[--top];}
  int  empty() {return !top;}
};
```

Zauważmy, że na stosie przechowujemy kopie.

Nuda...

- Dla każdego typu musimy definiować nową klasę...
- ...to nudne
- Możemy spróbować rozwiązać ten problem używając dziedziczenia, ale jak zobaczymy, to nie takie proste.
- Rozwiązaniem, które dziś poznamy są wzorce (templates).
- Spróbujmy jednak najpierw z dziedziczeniem:

```
// Podklasy tej klasy będą elementami stosu
struct EltStosu{
virtual ~EltStosu(){};
};
```

Stos elementów

```
class Stos {
  EltStosu** tab;
  unsigned size ;
  unsigned top ;
public:
  Stos(unsigned s=10)
    {tab=new EltStosu*[size=s];top=0;}
  ~Stos()
  void push(EltStosu el);
    {tab[top++]=new EltStosu(el);}
  EltStosu pop(void) ;
  int  empty() {return !top;}
};
```

Implementacja

Destruktor i `pop` wymagają więcej uwagi:

```
Stos::~Stos()
{
  while(top) delete tab[--top];
  delete[] tab;
}

EltStosu Stos::pop()
{
  EltStosu res(*tab[--top]);
  delete tab[top];
  return res;
}
```

Użycie

```
struct Liczba : EltStosu {
   int val;
   Liczba(int i):val(i){}
   operator int() {return val;}
};
main() {
   Stos s;
   s.push(Liczba(3));
   s.push(Liczba(5));
   while(!s.empty())
    cout << int(s.pop()) << endl ;
}
```

Niestety...

- ...wprawdzie wydaje się, że to całkiem sprytna implementacja, ale niestety nawet się nie skompiluje.
- Pierwszy problem stanowi wyrażenie `int(s.pop())`
- `s.pop()` jest typu `EltStosu`, a nie `Liczba` i konwersja do `int` nie jest dlań zdefiniowana.
- A może by tak użyć funkcji wirtualnych?

Jeszcze jedna heroiczna próba

```
struct EltStosu {
  virtual ~EltStosu(){}
  virtual void* operator()(){};//adres wartości
};
struct Liczba : EltStosu {
  int i;
  Liczba(int k):i(k){}
  void* operator()(){return (void*)&i;}
};
//...
// konwersja i dereferencja wskaźnika
    i = *(int*)s.pop()(); // Konwersja
```

Teraz program się kompiluje, ale nie działa :(

`void*` czyli “wiem co robię”

- Typ `void*` oznacza uniwersalny wskaźnik, bez precyzowania typu wskazywanych elementów.
- Nie można uzyskać elementu wskazywanego przez `void*`.
- Można za to dokonywać konwersji między `void*` a innym wskaźnikiem.
- Oznacza to powiedzenie kompilatorowi “wiem co robię”, ale czy rzeczywiście zawsze tak jest?

`void*` — przykłady

```
unsigned u = 0xCAFEBABE ; // szesnastkowo
unsigned* pu =  &u;
void* pv = (void*)pu ;
char* pc = (char*)pv ; //OK - pierwszy bajt u

cout << (*pc == char(0xbe)?"Small":"Big")
     << " endian" << endl ;
```

`void*` — przykłady

```
char c = 0;
unsigned u = 0xDEADBEEF;
char* pc = &c;
void* pv = (void*)pc ;
unsigned* pu = (unsigned*)pv ; // Śmieci (no, p

cout << *pu << endl; // ???
```

Problem z `push`

- Argument `push` jest przekazywany przez wartość (chcemy przechowywać kopie obiektów)
- . . . ale kopiowanie dokona się przez konstruktor kopiujący z klasy `EltStosu`.
- Na stosie będą zatem kopie tylko tych fragmentów obiektów, które pochodzą z `EltStosu`.
- Zmiana typu na

  ```
  void push(EltStosu&)
  ```

  . . . nie poprawia sytuacji.

Lepsze `push`

- Musimy zatem użyć wskaźników:

```
void push(EltStosu*){tab[top++]=el->kopia();
```

- oraz zdefiniować wirtualną metodę kopiującą:

```
struct EltStosu{
  virtual ~EltStosu(){}
  virtual EltStosu* kopia()=0;
};
struct Liczba : EltStosu {
  int i;
  Liczba(int k):i(k){}
  operator int() {return i;}
  EltStosu* kopia() {return new Liczba(i);}
};
```

Klasy abstrakcyjne

- Czasem definiujemy klasę, której obiekty nie będą tworzone.
- Będą tworzone jedynie obiekty jej podklas.

```
struct EltStosu{
  virtual ~EltStosu(){}
  virtual EltStosu* kopia()=0;
};
```

- `kopia()` jest tzw. *metodą czysto wirtualną*
- Co za tym idzie, `EltStosu` jest *klasą abstrakcyjną*.
- Nie można tworzyć obiektów takiej klasy, jedynie jej podklas (o ile nie są abstrakcyjne).

Poprawiamy `pop`

Metoda `pop` też jest zła — wynik jest typu `EltStosu`, zatem znów skopiuje się nieciekawy fragment. Na szczęście możemy ją dość prosto poprawić:

```
EltStosu* pop(void){return tab[--top];}
```

Przy okazji: zdefiniowanie metody `kopiuj()` jako czysto wirtualnej sprawiło, że `EltStosu` stał się klasą abstrakcyjną. Kompilator nie pozwoli nam na użycie jej konstruktora, więc błąd w `pop()` możemy szybko wykryć.
Refleksja: gdybyśmy od początku zdefiniowali `EltStosu` jako klasę abstrakcyjną, kompilator ostrzegłby nas od razu, że pierwotna koncepcja jest błędna.

Program główny

```
main()
{
   Stos s;
   s.push(&Liczba(3));//push kopiuje
   s.push(&Liczba(5));
   while(!s.empty()) {
      Liczba* l;
      l=(Liczba*)s.pop();
      cout << int(*l) << endl ;
      delete l;
   }
}
```

Podsumowanie

Wady:

- wkładając trzeba dokonać konwersji do podklasy `EltStosu`,
- dla każdego typu trzeba zdefiniować podklasę `EltStosu`,
- trzeba w niej zdefiniować konstruktor, operator konwersji i metodę `kopia()`,
- trzeba dokonywać rzutowań przy pobieraniu (niewygodne, a przede wszystkim niebezpieczne)
- trzeba pamiętać o usuwaniu pamięci po pobranych obiektach.

Wzorce (szablony)

- Wzorce są narzędziem umożliwiającym parametryzowanie struktur danych.
- Wzorzec klasy specyfikuje jak można konstruowac poszczególne klasy.
- Deklarację wzorca poprzedzamy słowem `template`, po którym w nawiasach kątowych podajemy parametry wzorca.
- Typowy parametr wzorca deklarujemy używając słowa `class` a potem nazwy parametru.
- Parametr faktyczny nie musi być klasą (może być w zasadzie dowolnym typem)

Wzorzec stosu

```
template <class T>
class Stos {
  T* tab;
  unsigned size ;
  unsigned top ;
public:
  Stos(unsigned s=10){tab=new T[s];top=0;}
  ~Stos(){delete[] tab;}

  void push(T el){tab[top++]=el;}
  T pop(void) {return tab[--top];}
  int  empty() {return !top;}
};
```

Używanie szablonów

Użycie wzorca polega na zapisaniu jego nazwy wraz z odpowiednimi parametrami (ujętymi w nawiasy kątowe). Takie użycie wzorca może wystąpić wszędzie tam, gdzie może wystąpić typ.

```
main()
{
   Stos<int>s;
   s.push(3);
   s.push(5);
   while(!s.empty()) {
      cout << s.pop() << endl ;
   }
}
```

Parametry rozmiaru

Oprócz typów parametrem wzorca może być wartość typu całkowitego (zwykle używana jako rozmiar:

```
template <class T, unsigned size>
class Stos {
  T* tab;
  unsigned top ;
public:
  Stos(){tab=new T[size];top=0;}
  ~Stos(){delete[] tab;}
  void push(T el){tab[top++]=el;}
  T pop(void) {return tab[--top];}
  int  empty() {return !top;}
};
Stos<int,20> s;
```

Parametry domyślne

Podobnie jak dla funkcji, dla wzorców możemy podawać domyślne wartości parametrów, np.

```
template <class T=int, unsigned size=20>
class Stos { ...
```

Przy takiej definicji

```
Stos<>
```

Znaczy tyle co

```
Stos<int,20> s;
```

Wzorce a zgodność typów

Dla każdych wartości parametrów tworzona jest nowa klasa. Co za tym idzie,

```
Stos<int,10>
```

oraz

```
Stos<int,20>
```

to dwie **niezwiązane ze sobą** klasy.

Szablony funkcji

Oprócz wzorców klas można tworzyć również wzorce funkcji globalnych. Oto deklaracja uniwersalnej funkcji maksimum:

```
template<class T>
T maks(T a,T b) // NB "max" już jest
{
  return (a>b?a:b);
}
```

i przykłady jej użycia

```
cout <<maks(24,42) << endl;
cout <<maks(string("Jacek"),string("Placek"))
     <<endl;
```