1 Pole

Jedna se o nejzákladnější a nejužitečnější datovou strukturu, se kterou se v programovani potkavame. V C se pole ovšem daji používat jen ve statických případech, kdy velikost můžeme stanovit ještě při kompilaci. Ovšem, i v připadech, kdy to bude dostačující, tak budeme li chtit předat pole do funkce, tak buď musime definovat onu funkci pro konkretni velikost pole, nebo využit ukazatele. První případ je použitelný jen zřídka, jelikož nepřináší abstrakci, kterou by jsme intuitivně chtěli, museli by jsme pro různé velikosti poli psat různé funkce.

To řeší předaní pole přes pointer a, po případě, ukončením pole nějakým specifickým symbolem, nebo předáním doplňujícího parametru délky. V takovém to připadě se však ztraci jaka koliv vyhoda vybraného datového typu, a vlastně i smysl onoho. Dostavame méně explicitní kod, ktery je více nadolný na chyby, jelikož compile time informaci, ktera se poji jen k jedne proměnné, rozvadime do dvou runtime.

Stanovíme tedy požadavky pro naše pole. Mělo by byt využitelné ve funkcich bez ztraty identity a při tom byt implicitním ukazatelem na počatek svých dat pří přiřazeni do pointeru. Navíc, rozšířít typ i na dynamické pole a dynamické pole variablní délky.

1.1 Délka pole

Zavedeme získatelnou délku pole.

```
int[8] arr;
arr.length; // vrati delku pole, tedy 8
```

Při předani pole do funce se tedy předa ukazatel na data a jako skryty parametr velikost pole. Pro připady, kdy neni potřeba předavat velikost, se muže použit pointer a implicitní přetypovaní.

1.2 Typy poli

1.2.1 Pole konstantní compile-time známé délky

Jedna se o pole analogické tomu, co je v C. Tedy

```
int[8] arr;
```

By vytvořilo pole o delce 8. Spočtená délka by se vždy dosazovala compile-time, realna proměnna pro jeji uchovani by neexistovala.

1.2.2 Pole konstantní run-time známé délky

Jedna se o analogii alokace ukazatele v C, ktery by byl využivany jako pole.

```
int* arr = malloc(sizeof(int) * 8);
```

Tedy

```
int[const] arr = alloc int[8];
```

By alokovalo pole o delce 8 na heapu a vytvořilo by proměnnou pro uložení délky.

1.2.3 Array List

Protože se array list často využiva, bylo rozhodnuto ho implicitně přidat do jazyka v ramci poli. Jedna se pouze o dynamické pole s automatickou realokaci při přidani prvku mimo rozsah Tedy int[] arr = alloc int[8]; By alokovalo pole o delce 8 na heapu a vytvořilo by proměnnou pro uložení délky.

1.3 Prace s polem

Protože přinašíme rozdělení pole od ukazatele, tak bych s touto myšlenkou pokračoval dal a vnímal pole jako definici prvku stejných vlastnosti, jako něco, co určuje chování pro všechny jeho prvky v jdnom mistě. Vyjdeme li z C, tak tato myšlenka je zde.

```
int arr[8];
```

Kde my vytvoříme kontejner pro 8 proměnných arr[0] .. arr[7] o datovém typu int, ktery specifikujeme jednou. Proměnná arr je však konstantní, jelikož neni zřejmé, jestli např.

```
int arr1[8];
int arr2[8];
arr1 = arr2;
```

ma přiřadit všechny prvky arr2 do arr1, nebo přepsat ukazatel arr1 na ukazatel arr2. To plyne z toho, že vlastně pole jako takové v C by se dalo říct,

že neexistuje. Jedna se fakticky vždy jen o ukazatel posypany syntaktickým cukrem a v takovém to připadě by, jaka koliv z předchozích operaci, by mohla byt vnímana jako hidden flow. My ovšem rozdělujeme mezi ukazatelem a polem na urovní typu. Tedy mužeme rozdělít chování ukazatele a pole a praci s polem vnímat vskutku jako praci se všemi proměnnými naraz, aniž by jsme něčemu ubližili.

2 String

V C string literaly jsou pouze hezčí verzí zapsani pole constantních charu. Vcelku, i když je to primitivní, tak zcela postačujicí. Problemem je zde 'absence' pole jako typu, jak již bylo zmiňovano[]. Tedy vlastně se s každym stringem pracuje jako s pointrem. Jelikož my vnímame pole jinak, tak mužeme rozvinout možnosti pole, aby nam umožňovaly ve vysledku lehčí praci i se stringy. Samotný typ pro string existovat nebude, ale bude jen podpora string literalu, ktery se při kompilaci rozloží na pole.

2.1 UTF-8

Bylo by vhodné rozšířít podporu literalu z ASCII na jiné kodovaní, aby byla jednoduchá cesta s připadnou jednoduchou manipulaci se složitějšími symboly. Jako takové kodovaní bych volil utf8, protože je kompatibilní s ascii, jeho blokem je byte, tedy neni zavísle na edianech a je velmi rozšířene. Jako nejlepší možnost bych viděl zadani literalu v utf8 a compile-time vyhodnocení nevětší delky potřebné pro uložení jednoho symbolu, a nasledně konverzi na pole intu o patřičné velikosti, kde každy element bude samostatnym symbolem zakodovaným v utf8.

2.2 Operace

Jako jediné konvenční operace nad stringy ktere by se měly integrovat do syntaxe, bych viděl concatinaci a slice. Ostatní operace by už měly byt obsažené v standardní knihovně.

2.2.1 Spojení

Níc nového bych nevymyšlel, a použil operator .. jako v Lua.

```
int[] str1 = "Hello";
int[] str1 = "World";
int[] str3 = str1 .. " " .. str2;
```

Implementace by však nesměl obsahovat žadnou alokaci pamětí, bylo by to zavadějicí. Jelikož délky dynamickych poli jsou jejich 'současti', šlo by to využivat pro alokavana pole. Ovšem, samotná, opět, nesmi obsahovat alokace, tedy připadné vysledné pole by se muselo samostatně standardně alokavt prostředky jazyka.

```
u8[const] arrC3 = alloc [] : arrA .. arrB;
```

...

2.2.2 Slice

...

2.3 Namespace

Zručný nástroj k organizací kodu. Umožňuje zhlukovat proměnné pod jedním společným názvem, který je rozlišitelný parsrem. Na rozdíl od použití identifikujících prefixu / postfixu v názvech je strukturním celkem z hlediska nástrojů operujících s kódem (např LSP). Umožňuje také při kompilaci hromadně pracovat s definovanými vevnitř proměnnými, a tedy se dá dobře využívat i pro import export části kódu (např python import foo; import from foo x;).

V našem případě by namespacem byl jednoduše pojmenovaný scope.

```
namespace Foo {
x;
}
```

K přístupu by se použila syntaxe z C++ Foo::x;

2.4 Import

Nejhorší části C jsou hlavičkové soubory a s nimi související systém importu. Hlavní nevýhodou kterého je duplicita definic. Slouží však k dobrému úmyslů, k izolaci implementací a definici rozhraní.

My teda budeme chtít tuto myšlenku ponechat a rozvít.

Základním celkem bude sobor. Všechny importy však budou probíhat v rámci AST, každý soubor by tedy měl být samostatně parsovatelným celkem. Prvně ošetříme možnost přímého importu souboru. import filename

Protože se v importovaném souboru mohou vyskytnout stejné názvy proměnných, chceme mít možnost zabalit ho do namespacu.

```
import filename as namespace Foo
```

Pak se nám vytvoří namespace Foo a kořen souboru se přidá jako jediný prvek do něj.

Obdobně by jsem mohli využít onoho syntetického konstruktu k implementaci jiných zabalení souboru. Např

```
import filename as scope
import filename as fcn foo
```

apod.

Dále, samozřejmě, budeme chtít umět vybrat patřičné namespace ze souboru (Popřípadě i identifikátory).

3 Function Overloading

I když se jedna o implicitní konstrukt, ktery skryva ... bla bla bla ... tak přinaši, z mého hlediska jednu zasadní věc, zjednošuená jmena funkci. Tedy, zamisto vepisovani datového typu do jmena funkce, mužeme jen uvest jeji činnost. To zjednodušuje vnimani samotného programu, jelikož při praci s vlastnimi datovymi typy, ktere definuji složité objekty, jmena funkci budou už znatelnou zatíží, oproti např. maxi, maxf, maxu, kde mužeme +- vydedukovat typ očekavane proměnné, se tak jednoduše nevystačime. Navíc jmena samotných funkci s použitím postfixu/prefixu, které si zvolime pro identifikaci, nebudou samostatnými celky z hlediska nastroju pracujicich s kodem, tedy v zakladu samotným kompilatorem a např LSP. Tedy nebude se moct nad nimi provadět žadna kontorla, tedy např kontorla překlepu, stíženy refaktoring, horší napověda a analyza atd..

Tedy, opět, z mého hlediska, je lepší ho mít, než nemit. Zbyva rozhodnout, zda povolit implicitni overloading, tedy jestli

```
foo(int x);
```

pro jiny datový typ musi byt explicitní cast

```
foo((int)1.0);
```

Zde však dochazi k zabavnému jevu. My explicitně vepisujeme datový typ, čímž identifikujeme funkci, ovšem o tom, jestli potřebná varianta existuje se dozvíme buď z LSP, v tomto připadě je cast jen z hlediska informace navíc (porovnavame li s implicitnim overloadingem), nebo při kompilaci, což už je trochu pozdě. Tedy jediny k čemu to muže sloužit je jako assert, kdy my víme, že chceme jit do konkretní varianty oné funkce, a pokud neexistuje, tak dostaneme error. Ovšem to budeme muset specifikovat u každého volani overloaded funkce, což se pře s tím, že chceme overloading hlavně z duvodu zjednodušené jmeńné stopy v kodu (nemluvě o tom, že vlastně mame tutež informaci dva x v kodu, jednou při definici, po druhy při volání). Bylo by tedy vhodné mit implicitní overloading, ale s opci v jistých připadech specifikovat konkretní požadovaný datový typ. Zavedeme tedy přisloušnou symboliku

```
foo!();
```

3.1 Implementace

V C++ implementuji nasledovně bla bla bla ... `https://en.cppreference.com/w/cpp/language/overload_resolution`

My budeme postupovat obdobně. Pro jmeno volané funkce najdeme všechny funckce se stejnymi jmeny a v odpovidajicím scope. Uložime do pole, kde v každem chlivečku bude struktura odkazujici se na funkci a doplňujici připadné informace popisujcí schodu. Pro zatím, neuvažujeme li polymorfismus, genericitu atd... si postačíme jen s jednou jedinou proměnnou typu int určujicí podobnost funcke našemu vzoru z volaní.

Budeme prochazet ono pole postupně funcki po funcki a buď je vyřazovat, nebo sestavovat skore podobnosti. Nakonci vybereme funkci s největším skore. Chyba nastane pokud budou dvě a vice stejná maximalní score, nebo nezbude žadna funkce se scorem.