Cubic Splines for bare-metal

Generováno programem Doxygen 1.8.17

Kapitola 1

Kubické splajny aneb Výhody C++ proti čistému C v bare-metal.

Na jednoduchých příkladech se zde pokusím ukázat, že použít jazyk C++ může být o něco efektivnější než psát to v čistém C. Většína "vychytávek" pochází až z moderního rozšíření C++11, použity jsou i konstrukce C++14. Kód bude kompletní příklad pro architekturu STM32F051, sice nedělá nic užitečného, ale měl by alespoň fungovat. No nic užitečného. Je to celkem kompletní příklad jak aproximovat nelinearitu K termočlánku (nebo jiné věci) kubickými splajny, využít by se to dalo, ale jak dále poznamenávám, trojčlenku zná každý, inverzi pásové matice dokáže málokdo, i když pokud tohle máme hotové, pak proti lineární aproximaci přibudou v jednom výpočtu jen 2x násobení a 2x součet, takže nic moc navíc.

Samotného mě překvapilo, co všechno z C++ a co není v čistém C obsaženo lze v takto krátkém příkladu využít. Určitě by se ještě něco našlo, časem sem možná ještě něco přidám, ale kromě tohoto zde ještě z kontextu vyplynulo, že gcc toolchain, používaný většinou bare-metal komunity není to jediné, co lze z GNU překladačů použít. Kdyby někdo hodně chtěl vyzdvihnout přednosti překladače clang, jen těžko by hledal vhodnější příklad. Velikost agresívně optimalizovaného kódu je pro Cortex-M0 tohoto projektu následující

```
size clang version 10.0.0 -Oz :
   text    data     bss     dec     hex filename
   4164       4     588    4756    1294 example.elf
size arm-none-eabi-gcc version 9.2.1 -Os :
   text    data     bss     dec     hex filename
  11304      44     848   12196    2fa4 example.elf
```

Takže pokud jste si mysleli, stejně jako já, že gcc produkuje efektivnější kód než clang, vězte, že tomu tak vždy být nemusí. Přiznávám, že to ale byl tak trochu úmysl. Nicméně, celý ten clang toolchain začíná dávat smysl, funguje i ld.lld, jen je trochu problém s knihovnami, ale lepší se to a doprovodné nástroje (scan-build) se docela hodí i pro gcc.

Kromě cílového procesoru STM32F051 to lze přeložit i jako nativní kód, zdrojáky kde je nějaký rozdíl jsou v odlišných adresářích. Je to sice o dost víc práce, ale zapatlávat to celé těmi ifdef, ve kterých se nakonec nikdo nevyzná se mi opravdu nechtělo.

Tento text není a nebude učebnice jazyka C++. Je to jen pár neúplných poznámek o jeho využití v bare-metal.

1.1 Obsah.

1. Typová kontrola.
2. Zapouzdření.
3. Jmenné prostory.

4. Konstruktory a destruktory.
5. Reference.
6. Virtuální metody.
7. Přetěžování funkcí, operátory.
8. Šablony a STL.
9. Constexpr výrazy.
10. Lambda výrazy.
11. Ostatní dekorace.
    (a) Raw String Literal
    (b) User-defined literals
    (c) Range-based for
12. Závěr.

1.2 Typová kontrola.

Lepší než v C, eliminuje některé chyby, ale není úplně striktní jako je tomu třeba v jazyce rust. I zde probíhají automatické konverze ze signed na unsigned (a naopak), což může být zdroj chyb. Dále je potenciálně nebezpečné zúžení celočíselného typu - viz. havárie rakety Ariane 5. Překladač na to obvykle upozorní, ale i zde je potřeba být velmi obezřetný. Stejně jako v čistém C je dobré používat celočíselné typy s definovanou šířkou z hlavičky stdint.h. Explicitní přetypování funguje se stejnou syntaxí jako v čistém C, je však lepší používat static_cast, const_cast, reinterpret_cast. Pak je na první pohled patrné, že programátor snad ví, co dělá, dá se to lépe dohledat.

Jde to dotáhnout o něco dále, zapouzdřit si vhodný celočíselný typ do extra třídy (nazvěme jí např. FIX::real), vytvořit si potřebné konstruktory, přetížit všechny potřebné operátory (ale je jich fakt moc) a případná přetečení ošetřit přímo na místě. Eliminují se tak automatické konverze. Oni to takhle měli patrně u té Ariane 5 uděláno, ale pro větší efektivitu kontroly prostě vypnuli. Programování bývá o kompromisu, ale nesmí se to přehánět.

S touto problematikou souvisí i používání statických inline (příp. constexpr) funkcí (výrazů) místo C-čkových maker. Je to typově bezpečnější a moderní překladače si s tím poradí, aniž by generovaly nějaký kód navíc.

Takže když vidíte v popise periferií toto

```
static USART1_Type      & USART2      = * reinterpret_cast<USART1_Type      * const> (0x40004400);
static USART1_Type      & USART3      = * reinterpret_cast<USART1_Type      * const> (0x40004800);
static USART1_Type      & USART4      = * reinterpret_cast<USART1_Type      * const> (0x40004c00);
static USART1_Type      & USART5      = * reinterpret_cast<USART1_Type      * const> (0x40005000);
static I2C1_Type        & I2C1        = * reinterpret_cast<I2C1_Type        * const> (0x40005400);
static I2C1_Type        & I2C2        = * reinterpret_cast<I2C1_Type        * const> (0x40005800);
```

neděste se toho. To, že je místo hvězdičky ampersand značí jen, že místo ukazetele používám odkaz (Reference.). Lépe se to pak píše a není to nic proti ničemu. V C-čku se na tohle používaly makra i když by šly také použít statické konstanty, ale zřejmě to je jen síla zvyku. *Pokud je vám divné, že je tam něco jako USART1_Type nebo I2C1_Type, když by to šlo i bez té jedničky, je to tak proto, že jak je hlavička* `generována` *ze* `SVD` *souboru, probíhá zároveň kontrola, zda jsou u periferie stejného typu stejné registry a v nich stejné i bity. Pokud tomu tak je, pak je teprve periferie považována za odvozenou a přejmenována podle původní. A nebyla už síla to nakonec umravnit. Problém byl v tom, že ST používalo timery stejného typu i když v některých chybí mnohé bity. To mi zkrátka vadilo.*

1.3 Zapouzdření.

Je nejdůležitějším rysem jazyka C++. Tato myšlenka (spolu se jmennými prostory) umožnila trochu umravnit velké projekty, je šikovné mít pohromadě data i funkce, které s nimi manipulují. Ten János Neumann, známý jako John von Neumann (ano původem Maďar) byl hodně chytrý člověk, kromě počítačů dal kompletně dohromady i základy kvantové mechaniky. Nebudu to příliš rozebírat, v kódu je zapouzdřeno do tříd nebo struktur vše, co má jen trochu smysl. Opět - negeneruje to žádný kód navíc.

Kromě přehlednosti to umožňuje i snadnější přenositelnost - zdrojový kód lze jednoduše přenést do jiného projektu, v zásadě není nutné v tom dělat nějaké úpravy. Sice se výkonný kód dostává do hlavičkových souborů, což je v čistém C považováno za nežádoucí, ale na druhou stranu metody definované už v hlavičce jsou automaticky inline, kód se tak o něco zrychlí.

1.4 Jmenné prostory.

V takto malém projektu je poměrně těžké najít smysluplné využití. Tady se pomocí jmenných prostorů pokusím trochu suplovat preprocesor. Výpočty byly původně napsány obecně pro aritmetický typ real. Co tento typ představuje je dodatečně definováno v hlavičkách real_fix.h a real_flt.h a prostým include jedné z nich se dalo vybrat, zda to bude řešeno v pevné nebo pohyblivé řádové čárce. V C++ lze celý obsah hlavičky obalit jmenným prostorem a vložit je do zdrojáků obě. Zůstane jediný bod ve kterém je možné udělat výběr :

```
#include "real_fix.h
#include "real_flt.h
// Zde je možné vybrat si, zda se to bude počítat v pevné
// nebo pohyblivé řádové čárce. Pro Cortex-M0 stačí pevná, je to de facto int32_t.
   using namespace FIX;
// using namespace FLT;
```

Jasně, nic moc to nepřináší, ale funguje to. Jmenné prostory oceníme v knihovnách a rozsáhlých projektech, tady je to opravdu zbytečné, něco jsem sem dát musel, protože jmenné prostory v čistém C nejsou a tohle má alespoň nějaký smysl. A takto se to normálně nepoužívá.

1.5 Konstruktory a destruktory.

Patří k zapouzdření tak nějak přirozeně. V čistém C docházelo k neočekávaným chybám tím, že programátor prostě někde zapomněl inicializovat nějakou proměnnou. Prostě očekával, že v paměti je nula a ona tam z nějakého důvodu nebyla. To je typické pro zásobník - v něm zůstává vždy nějaký obsah z předchozí činnosti. Právě tohle by měl řešit konstruktor.

Další problém byl při alokaci paměti na haldě. V čistém C, pokud je projekt trochu složitější je dost velký problém uhlídat, jestli je alokovaná paměť uvolňována vždy a ve správný okamžik (otázka vlastnictví objektu a jeho doby života). Konstruktory a destruktory tohle řeší sice jen částečně - pokus o definitivní řešení přišel až s jazyky jako je java, python atd., kde se používá tzv. garbage collector (neplést s uklízením nepotřebých sekcí při linkování), ale to přináší jen další problémy - ale v bare-metal se s tím nemá cenu moc zabývat. Alokace na haldě v bare-metal nemá příliš velké využití (alespoň já se snažím tomu vyhnout jak čert kříží), na druhou stranu v čistém C zase nikdo moc nevytváří složitější datové struktury lokálně. Původně to ani moc nešlo, nějak se to divně obchází funkcí alloca(), modernější C-čko už umí alokovat pole na zásobníku podobně jako C++. Je to k diskusi, ale pokud máme RAM opravdu málo, tak se s tím prostě moc dělat nedá. Jediné co jde, je použít stejný úsek paměti v různý čas. Zásobník je na to stejně dobrý jako halda, ale někdy to prostě nejde nebo se tím problém hodně zesložití. Halda je dobrá, pokud vytváříme objekt, jehož velikost závisí na příchozích datech. To nebývá často a přináší to problémy s fragmentací a nedeterministickým časem přístupu. To samozřejmě není problém jazyka, ale bare-metal jako takového. Halda je relevantní v problémech hromadného zpracování dat, databází a pod., zde většinou pracujeme v reálném čase, kdy je spíš potřeba okamžité reakce na nějakou změnu vstupu a objemy zpracovávaných dat nejsou velké.

Co je trochu problém v bare-metal to jsou konstruktory statických tříd. Naštěstí je překladač umísťuje na zcela určité místo, takže pak stačí (ještě před voláním main()) udělat něco jako

```
extern void (*__init_array_start)(); // definováno v linker skriptu
extern void (*__init_array_end)  (); // definováno v linker skriptu
void static_init() {
  void (**p)();
  for (p = &__init_array_start; p < &__init_array_end; p++) (*p)();
}
```

Vypadá to dost šíleně, ale funguje to a bez toho to opravdu nejde.

Inicializace periferií v (noexcept) konstruktoru ? Je to též otázka k diskusi, já to takto používám a funguje to. Dokonce lze periferii v destruktoru přes RCC zresetovat a odstavit. To se hodí pro nějaké sdílené funkce nebo v low power režimech.

1.6 Reference.

Neboli odkazy jsou také něco, co není v čistém C a je to dost užitečné. A není to jen takový "lepší" zápis ukazatele. Začátečníka v C++ to může mást, lze si na to však zvyknout a používání se vyplatí. Překladač si tak nějak hlídá aby nebylo možné vytvářet reference na objekt, který neexistuje (skončila mu doba života), na rozdíl od ukazatelů, kde je pak možné přes (nenulový) ukazatel například volat metodu objektu, který je už dávno mimo paměť. Což zamo-zřejmě skončí fatální chybou. Bez referencí by se neobešly kopírovací konstruktory. Ale nechtěl bych to rozebírat do detailů, kterým stejně úplně nerozumím, přečtěte si Bruce Eckela, stačí si zapamatovat, že se s tím líp pracuje a také to o něco lépe funguje. Je to prostě něco jako konstantní ukazatel, který se automaticky dereferencuje. Kód tím také nijak neroste. A možná je to na pochopení lepší než ty ukazatele. Údajně to pochází z jazyka Algol nebo Pascal. Používám to v programech často a zdá se, že to spokojeně funguje.

1.7 Virtuální metody.

Umožňují vytvářet programová rozhraní bez zásahu do původního kódu. V čistém C se používají callback funkce, virtuální metody jsou o něco přehlednější, ale pro někoho příliš abstraktní. Dost špatně se to vysvětluje, ale zkusme to. Mějme protokolový stack a představme si, že hardware je jaksi "naspodu", je to třída Usart a její instance usart. Nad ní je jediná další vrstva, třída Print, resp. její instance console. Obě tyto třídy dědí z třídy BaseLayer a pro jednoduchost si představme, že data jsou vytvářena (výpočtem) v hlavní smyčce programu třídou Print, resp. její metodou BlockDown, všechno co je potřeba lze celkem snadno do ní svést.

První co uděláme, je spojení instancí jmenovaných tříd

```
  console += usart;                       // abstraktní zřetězení pomocí třídy BaseLayer
```

funguje to pak takto

```
    virtual BaseLayer & operator += (BaseLayer & bl) {
      bl.setUp (this);  // ta spodní bude volat při Up tuto třídu
      setDown  (& bl);  // a tato třída bude volat při Down tu spodní
      return * this;
    };
  protected:
    void setUp   (BaseLayer * p) { pUp   = p; };
    void setDown (BaseLayer * p) { pDown = p; };
  private:
    // Ono to je vlastně oboustranně vázaný spojový seznam.
    BaseLayer * pUp;
    BaseLayer * pDown;
```

potřebujeme dávat data "dolů", takže budeme využívat jen metodu Down() bázové třídy BaseLayer

```
    virtual uint32_t    Down (const char * data, const uint32_t len) {
      if (pDown) return pDown->Down (data, len);
      return len;
    };
```

z toho by mělo už být vidět, že nahoře, tedy ve třídě Print metodu Down() ani přetěžovat nemusíme, stačí jí jen zavolat

```
// Výstup blokujeme podle toho, co se vrací ze spodní vrstvy
uint32_t Print::BlockDown (const char* buf, uint32_t len) {
  uint32_t n, ofs = 0, req = len;
  for (;;) {
    // spodní vrstva může vrátit i nulu, pokud je FIFO plné
    n = BaseLayer::Down (buf + ofs, req);
    ofs += n;   // Posuneme ukazatel
    req -= n;   // Zmenšíme další požadavek
    if (!req) break;
    sleep();    // A klidně můžeme spát
  }
  return ofs;
}
```

co přetížit musíme je Down() ve spodní vrstvě Usart

```
uint32_t Usart::Down (const char * data, const uint32_t len) {
  uint32_t res;                 // výsledek, musí žít i po ukončení smyčky
  for (res=0; res<len; res++) if (!tx_ring.Write(data[res])) break;
  USART1.CR1.B.TCIE = SET;      // po povolení přerušení okamžitě přeruší
  return res;
}
```

kde jak je vidět cpeme jen příchozí data do fronty (FIFO) tx_ring (proto vlastně metody Up() i Down() nemohou být konstantní). Jak se data v přerušení z fronty vybírají a jak udělat frontu atomicky, tím se tu zaobírat nebudu.

Z toho celého povídání se zdá, že pro čistého C-čkaře to musí být na blázinec, ale lze si na to zvyknout, opravdu na tom nic není, tohle jednou napíšete a pak to jen používáte. A když to takhle vyzobu ze zdrojáků aby se to důležité vešlo na jednu obrazovku, je vidět, že je to opravdu jen pár velmi jednoduchých funkcí, bohužel jsou však dost složitě provázány a dohledávat to ve zdrojácích není snadné. V C-čku by to šlo napsat jednodušeji tak, že by se to dalo úplně všechno do jedné funkce - callbacku, hodně by tím však utrpěla možnost znovupoužití tohoto kódu. Opět není generován žádný zbytečný kód navíc, uvádí se, že pokud takto používáme dědičnost a virtuální metody, překladač vyneruje navíc tabulku VTABLE, ale pokud jsou metody takhle jednoduché a definované přímo v hlavičce, jsou inline a nepřekáží. Možná tam nějaká tabulka VTABLE bude, možná nebude, protože překladač v takto jednoduchém případě dynamickou vazbu nepotřebuje, celá problematika do všech detailů jak to přesně funguje je složitá, ale celkem není potřeba tomu rozumět až na dřeň. Stejně je to tak se vším, málokdo chápe matematický aparát kvantové elektrodynamiky do té míry aby dokázal posoudit zda symetrie U1(loc) opravdu generuje zákon zachování elektrického náboje, ale učíme se už na střední škole, že to tak je a není důvod tomu nevěřit. A že je Země kulatá, je též jen otázka víry.

Co by bylo trochu problematické je chod dat v opačném směru, příjem dat Usartem. Metoda Up() by se pak volala v přerušení a v něm by pak probíhala celá obsluha protokolového stacku. Ne, že by to nešlo, já mám mnoho prográmků, kdy se celá práce děje v přerušení, v main smyčce se jen uspává jádro, ale tam se většinou používá DMA. Prostě musí se nad tím trochu přemýšlet.

Nakonec je třeba se krátce zmínit o abstraktních bázových třídách. V projektu to použito není a udělat na to jednoduchý a přehledný příklad prostě nejde. Pokud napíšeme něco jako

```
struct AbstactInterface {
  virtual bool Send (const char * data, const int len) = 0;
};
```

máme abstraktní bázovou třídu (strukturu AbstactInterface, je to jedno). Zde obsahuje jen jedinou metodu Send(), důležité je to = 0 na konci. To značí čistou virtuální metodu (pure virtual) a pokud třída obsahuje jen jedinou takovou metodu, je překladačem považována za abstraktní a nemůžeme pak vytvořit její instanci. Pokud takovou třídu zdědíme, pak musíme čistě abstraktní metodu přetížit, tj. vytvořit její tělo. V čistém C je to něco jako callback funkce - tady lze zapomenout tento callback nastavit, což pak končí chybou. V C++ to nejde, překladač nás s tím vyhodí. Je to hodně zjednodušeně řečeno, ale podrobnosti nejsou zase tak podstatné. Opět bych odkázal na Eckela, jeho dvojdílná bichle "Myslíme v jazyce C++" je dostatečně podrobná, aby každého přešla chuť se C++ naučit.

1.8 Přetěžování funkcí, operátory.

Užitečné, hlavně pro rekurzívní volání. V čistém C nemůžou mít funkce stejná jména i když mají jiné parametry. V C++ to možné je, překladač si ta jména interně "zmrší" (mangle) právě podle těchto parametrů. No a protože operátor je jen odlišná syntaxe pro funkci, funguje to i pro operátory, kde se to používá snad nejvíc.

Příklad.

```
    Print &    operator « (const char * str);
    Print &    operator « (const int    num);
    Print &    operator « (const real & num);
    Print &    operator « (const char   c);
    Print &    operator « (const PrintBases num);
```

Operátor << je snad nejčastěji přetěžovaný, naznačuje něco jako "výstup", ale v těchto zdrojácích se běžně přetěžují i ostatní operátory - třeba pro aritmetické operace. Třída real tedy může simulovat aritmetiku v pevné řádové čárce nebo může fungovat jako běžné float číslo. Za zmínku stojí, že operátory běžně vracejí odkaz na třídu (resp. instanci), ze které jsou volány (this), což umožňuje jejich řetězení (to je při zápisu aritmetických výrazů běžné).

1.9 Šablony a STL.

Také dost usnadní práci, nemusí se přepisovat kód pro každou blbost, v C samozřejmě nic takového není. Tady je typický příklad třída FIFO (dbus_w_t je nějaký celočíselný typ, který daná architektura umí uložit atomicky)

```
template<typename T, const dbus_w_t M = 64> class FIFO {
  T m_data [M];
  volatile dbus_w_t m_head;
  volatile dbus_w_t m_tail;
  // ...
```

která má 2 parametry - typ T ukládaného objektu a M (implicitně 64) celé číslo udávající počet ukládaných prvků. Tak je možné vytvářet instance staticky, není potřeba používat haldu. Za zmínku stojí konstruktor

```
    explicit constexpr FIFO<T,M> () noexcept {
      // pro 8-bit architekturu může být byte jako index poměrně malý
      static_assert (1ul « (8 * sizeof(dbus_w_t) - 1) >= M, "atomic type too small
);
      // a omezíme pro jistotu i delku buferu na nějakou rozumnou delku
      static_assert (isValidM (3, M), "M must be power of two in range <8,4096> or <8,128> for 8-bit data
       bus (AVR)
);
      m_head = 0;
      m_tail = 0;
    }
```

a v něm static_assert(). Výraz ve static_assert() musí být constexpr, je to kontrola za překladu. To také v C nejde. Při překladu to kontroluje, zda M není blbost a vcelku zadarmo. Atomičnost operací zde rozebírat nebudu, ale když už jsme u šablon nelze nezmínit STL, čili Standard Template Library. Ta není moc použitelná v bare-metal, prvky jako std::string, std::vector jsou nenažrané a používají haldu, ale třeba std::atomic na Cortex-M3/4 lze použít s výhodou, používají instrukce LDREX, STREX, takže nezastavují přerušení procesoru jak je to běžné třeba na AVR. Celé to prozkoumané nemám, ale jde použít např. std::sort() na druhou stranu třídit něco v tak malé RAM lze i jednodušeji.

V C by snad šlo použít makra ?

1.10 Constexpr výrazy.

Co jde spočítat při překladu, to spočítej a výsledek dosaď do výsledného kódu. Zatím to umí dobře clang (zřejmě kvůli LLVM struktuře), lze i velmi složité analytické funkce, dokonce tabulky do flash. Pro ARM je hezkou ukázkou výpočet koeficientů kubických splajnů, kde clang dá podstatně menší kód než gcc (o několik KiB). Je to jen tím, že gcc ten výpočet ponechá ve flash, počítá to za běhu, nikoli už při překladu.

```
    explicit constexpr SPLINE (const Pair * const p, const bool reverse = false) noexcept {
      double x [N], y [N];  // přeskupení dat - možnost inverzní funkce při reverse = true
      if (reverse) {for (int i=0; i<N; i++) { x[i] = p[i].y; y[i] = p[i].x; }}
      else         {for (int i=0; i<N; i++) { x[i] = p[i].x; y[i] = p[i].y; }}
      /* Tenhle příšerně složitý konstruktor je převzat ze stackoverflow.com (původní odkaz už zmizel)
       * Není to zas taková sranda - koeficienty se počítají řešením tridiagonální matice řádu N.
       * V těch indexech se snadno zabloudí, tohle kupodivu funguje (přirozené splajny) ač je to dost
       krátké.
       * Je zajímavé, že tohle najdete spíš ve Fortranu než v C, matematici jsou zřejmě hodně konzervativní.
       * */
      const int n = N - 1;
      double h [n];
      for (int i = 0; i < n; ++i) { h[i] = x[i+1]-x[i]; }
      double alpha [n]; alpha [0] = 0.0;
```

```
    for (int i = 1; i < n; ++i) {
      alpha[i] = 3.0 * (y[i+1] - y[i]) / h[i] - 3.0 * (y[i] - y[i-1]) / h[i-1];
    }
    double c [n+1], l [n+1], mu [n+1], z [n+1]; l [0] = 1.0;  mu[0] = 0.0;  z [0] = 0.0;
    for (int i = 1; i < n; ++i) {     // přímý chod
      l [i] = 2.0 * (x[i+1] - x[i-1]) - h[i-1] * mu[i-1];
      mu[i] = h[i] / l[i];
      z [i] = (alpha[i] - h[i-1] * z[i-1]) / l[i];
    }
    l[n] = 1.0;  z[n] = 0.0;  c[n] = 0.0;
    double b [n], d [n];
    for (int j = n-1; j >= 0; --j) {  // zpětný chod
      c[j] = z [j] - mu[j] * c[j+1];
      b[j] = (y[j+1] - y[j]) / h[j] - h[j] * (c[j+1] + 2*c[j]) / 3.0;
      d[j] = (c[j+1] - c[j]) / (3.0 * h[j]);
    }
    for (int i = 0; i < n; ++i) {     // závěrečný zápis koeficientů
      data[i].x = to_real(x[i]);   data[i].a = to_real(y[i]);
      data[i].b = to_real(b[i]);   data[i].c = to_real(c[i]); data[i].d = to_real(d[i]);
    }
  }
```

to je fakt už vyšší dívčí, celé je to v double kvůli přesnosti, počítá se inverze pásové matice protože aproximační polynomy na sebe nejen musí spojitě navazovat, ale výsledná funkce musí být i hladká. Tohle samozřejmě nikdo nepoužívá, je to tak složité, že pokud to někdo použít musí, tak si to předpočítá externí knihovnou (python je na to velmi dobrý, má knihovny na všechny možné interpolační funkce). Normální člověk použije lineární interpolaci, stačí mu na to obyčejná trojčlenka. Sice musí mít pro požadovanou přesnost řádově větší tabulku koeficientů, ale většinou to nevadí.

Celkem to není nic nového, i v čistém C, pokud použijeme trochu agresívnější optimalizaci, jednoduché funkce jejichž argumentem je číslo jako literál většinou vyprodukují při překladu rovnou výsledek. Tady je to jen dotaženo jen o něco dál. Obecně je dobré označit **úplně** všechno, co takto označit jen trochu jde jako const. Jednak to usnadní život překladači a pak pokud překladač někde vyhodí chybu, je jasně vidět, že jsme něco nedomysleli. Udržet v hlavě všechny vazby, které drží překladač je takřka nemožné.

1.11 Lambda výrazy.

Výhodou je, že to umí posbírat data z okolního kontextu a není tedy potřeba předávat tolik parametrů. V bare-metal jsem to použil pro nastavování jednotlivých bitů v bitovém poli metodou read-modify-write, je to přehledné a nepřináší to žádnou režii navíc. Ale jde to poměrně slušně udělat i v čistém C, je to jen o něco víc práce.

Příklad.

```
  USART1.CR1.modify([] (auto & r) -> auto {
    r.B.DEAT   = 1u;       // doba vybavení před start bitem - 16 ~= 1 bit, 0..31
    r.B.DEDT   = 1u;       // doba vybavení po stop bitu     - 16 ~= 1 bit, 0..31
    return r.R;
  });
```

a jak je to definováno (úplně na konci)

```
struct USART1_Type      {
  union CR1_DEF  {
    struct {
      __IO ONE_BIT    UE     :  1;
      __IO ONE_BIT    UESM   :  1;
      __IO ONE_BIT    RE     :  1;
      __IO ONE_BIT    TE     :  1;
      __IO ONE_BIT    IDLEIE :  1;
      __IO ONE_BIT    RXNEIE :  1;
      __IO ONE_BIT    TCIE   :  1;
      __IO ONE_BIT    TXEIE  :  1;
      __IO ONE_BIT    PEIE   :  1;
      __IO ONE_BIT    PS     :  1;
      __IO ONE_BIT    PCE    :  1;
      __IO ONE_BIT    WAKE   :  1;
      __IO ONE_BIT    M      :  1;
      __IO ONE_BIT    MME    :  1;
      __IO ONE_BIT    CMIE   :  1;
      __IO ONE_BIT    OVER8  :  1;
      __IO uint32_t   DEDT   :  5;
      __IO uint32_t   DEAT   :  5;
      __IO ONE_BIT    RTOIE  :  1;
      __IO ONE_BIT    EOBIE  :  1;
```

```
      __IO ONE_BIT    M1     :  1;
    } B;
    __IO uint32_t  R;
    explicit CR1_DEF () noexcept { R = 0x00000000u; }
    template<typename F> void setbit (F f) volatile {
      CR1_DEF r;
      R = f (r);
    }
    template<typename F> void modify (F f) volatile {
      CR1_DEF r; r.R = R;
      R = f (r);
    }
  };
```

Je dobré si všimnout, že je to šablona, takže takto generovaná hlavička pro popis periferií je v čistém C nepoužitelná. Šlo by to obejít (např. F jako ukazatel na funkci), ale ukázalo se, že je s tím víc komplikací než užitku, takže to zde zůstalo jako obecný "callable" objekt, což může být i třída s přetíženým operátorem (). Ani tahle konstrukce nemá žádný overhead proti tomu jak by to šlo napsat v čistém C. Toto bylo zvoleno záměrně - prvky pole jsou široké 5 bitů, takže by bylo dobré je vynulovat pomocí masky a pak teprve nastavit potřebné bity. Upřímně řečeno, používání bitových polí a unionů není úplně správně, přímočarý zápis v C jako read-modify-write celého "slova" je čistší, ale mně se osobně nelíbí. Je to dost guláš. Ale bitová pole závisí na endianitě a union může být závislý na překladači, takže je to vlastně hodně špatně. Nikdy jsem však nenarazil na problém. Endianita procesoru je předem známa a jak gcc tak clang se chovají vůči unionu poměrně slušně a předvídatelně.

Dovolím si malou poznámku k typu ONE_BIT. Je to výčet

```
enum ONE_BIT { RESET = 0, SET = 1 };
```

stejně jako v C. Jde tam tedy přiřadit pouze hodnoty SET nebo RESET. Pokud by zde bylo místo ONE_BIT uint32_t, nic špatného by se nestalo, ale šly by nastavit i hodnoty 0u nebo 1u. Takhle to kvůli typové kontrole nejde a je to tak dobře. ST výčty v SVD popise (asi z lenosti) moc nepoužívá a je to škoda. Je možné si něco dodatečně definovat jako třeba

```
typedef enum {
  USEHSI = 0, USEHSE, USEPLL
} SW_EN;
```

a pak použít

```
  switch (RCC.CFGR.B.SWS) {
    case USEHSI:  /* HSI used as system clock */
      SystemCoreClock = HSI_VALUE;
      break;
    case USEHSE:  /* HSE used as system clock */
      SystemCoreClock = HSE_VALUE;
      break;
    case USEPLL:  /* PLL used as system clock */
      /* Get PLL clock source and multiplication factor */
      pllmull = RCC.CFGR.B.PLLMUL + 2u;
      // ...
```

I když je RCC.CFGR.B.SWS typu uint32_t, není problém. V C++ lze definovat pro výčet i jeho typ, zde to není použito, protože v gcc mám zatím neopravený bug, který otravuje s pochybnými warningy, a ty nejdou vypnout. Je těžké říct jestli je to chyba nebo vlastnost. Nicméně používání výčtů je velmi užitečné, dost to omezí možnost přiřadit do proměnné úplnou blbost.

1.12 Ostatní dekorace.

1.12.1 Raw String Literal

velmi užitečné, v čistém C není (zřejmě pochází z pythonu).

Příklad.

```
static constexpr const char * Intro = R"-!-(
Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Curabitur sagittis hendrerit ante.
Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem. Aliquam erat
volutpat. Sed vel lectus. Donec odio tempus molestie, porttitor ut, iaculis quis, sem.
)-!-";
```

Fakticky se tohle dělalo podobně v čistém C pomocí assembleru a jeho direktivy .incbin "file.txt". Ale tohle se dá s výhodou použít naopak pro assembler vestavěný v C++ - asm (R"---(text)---");. Blok je pak daleko přehlednější, nemusí v něm být ty podivné escape sekvence.

1.12.2 User-defined literals

hodně divná věc, dají se s tím dělat celkem kouzla jako zapisovat čísla v libovolné číselné soustavě atd.

Příklad.

```
// Funkce s defaultním parametrem - to v čistém C nejde.
static void delay_ms (const unsigned ms = 500) {
  GlobalCounter = ms;
  while (GlobalCounter) {   // dekrementováno v přerušení od SysTick
    asm volatile ("wfi
");   // a proto můžeme při čekání uspat jádro
  }
}
// User-defined literal
static constexpr unsigned operator"
_s (const unsigned long long arg) {
  return 1000 * arg;        // pracujeme v ms, arg v s tedy zvětšíme 1000x
}
void LedBlinkTest () {
  for (;;) {
    led « true;            // rozsvit' (přetížený operátor « pro GpioClass)
    delay_ms ();            // použij defaultní argument (500 ms)
    led « false;           // zhasni
    delay_ms (2_s);         // použij user-defined literal
  }
}
```

Ukazuje jak převést sekundy na milisekundy, je to pitomost, ale podobně lze zapisovat třeba úhlové stupně v radiánech (a naopak). To se celkem hodí, je však nutné dodržet správný typ parametru - většinou to bývá nějaký dlouhý typ.

Kromě toho je v tomto příkladu ukázáno použití defaultního parametru, což se v kódu vyskytuje poměrně často, v čistém C toto není možné. Bohužel to v C++ není dotaženo do takové dokonalosti jako třeba v pythonu, je nutné být obezřetný.

1.12.3 Range-based for

proměnný počet parametrů, přetížené funkce print, jmenné prostory

Příklad - definice důležitých proměnných

```
// Naměřené hodnoty termočlánek K - pro dostatečnou přesnost stačí krok 20 °C
static constexpr Pair meassure[] = {
  {-20 , -757  }, // pár je vždy teplota ve °C a napětí článku v V
  { 0  , 0     },
  { 20 , 790   },
  { 40 , 1612  },
  { 80 , 3358  },
  { 120, 5228  },
  { 160, 7209  },
  { 220, 10362 },
  { 280, 13709 },
  { 320, 16032 },
  { 360, 18422 },
  { 380, 19641 },
  { 400, 20872 },
  { 420, 22110 },
};
static const SPLINE<array_size (meassure)> dcs (meassure, false);
static Print   console (DEC);
```

a pak to použijeme např. takto

```
namespace fmt {
  // Silně zjednodušeno, % jen naznačuje přítomnost parametru v seznamu
  constexpr void print (const char * fmt) {   // ukončení rekurze když už žádný parametr nezbyl
    for (;;) {
      const char c = *fmt++;
      if (c == '\0') break;
      console « c;
    }
  }
  template<typename First, typename ... Rest>
  constexpr void print (const char * fmt, const First & first, const Rest & ... rest) {
    for (;;) {
      const char c = *fmt++;
```

```
      if (c == '\0') break;    // pro jistotu, zde patrně zbytečné
      if (c == '%' ) { console « first; print (fmt, rest ...); break; }
      else console « c;
    }
  }
};
static void printCoeficients () {
  console « "Computed coeficients:
 « EOL;
  for (const auto & e: dcs) {    // Ukázka použití range based for
    fmt::print ("x = % : y = %, b = %, c = %, d = %\r\n
, e.x, e.a, e.b, e.c, e.d);
  }
}
```

funkce print jsou obaleny (naprosto zbytečně) jmenným prostorem fmt, jde to, chleba to nežere. Je to něco jako C-čkové printf(), výpisy jdou do console, proměnný počet argumentů je dělán jako šablona, postupné užití parametru je realizováno rekurzí. Typ parametru je při překladu znám, takže není nutné předepisovat typ ve formátovacím řetězci. A protože mě nezajímá ani délka výstupu (pro daný typ), stačí jen naznačit znakem procento, že se má sežrat další parametr. Ten přístup je proti C-čku prostě úplně jiný. Asi se tím nic neušetří, vypadá to blbě, rekurze žere zásobník (není úplně na ocase), zřejmě by to chtělo ještě jedno přetížení print(const char * fmt, const First & first), ale i tak to funguje a překladač nenadává. Je to prostě ukázka, že to jde.

Range-based for je lepší vychytávka, kterou v C nenajdete, ale v novějších jazycích je běžná. Je to určeno pro procházení pole dat aniž by bylo nutné používat nějaký index. Ale co si budeme povídat, většinou je stejně nějaký index potřeba. A pokud to opravdu chceme používat ve svých třídách, je k tomu potřeba ještě dodefinovat iterátory begin a end

```
    class iterator {
      const SplineSet * ptr;
      public:
        iterator(const SplineSet * _ptr) : ptr (_ptr) {}
        iterator  operator++ ()                                { ++ptr;   return * this;    }
        bool      operator!= (const iterator & other) const { return   ptr != other.ptr; }
        const SplineSet & operator*  ()               const { return * ptr;              }
    };
    iterator begin () const { return iterator (data        ); }
    iterator end   () const { return iterator (data + N - 1); }
```

což sice není žádná věda, ale zdržuje to.

1.13 Závěr.

To je zatím vše. Mělo by to mít nějakou licenci, ale jako obvykle napíšu - dělejte si s tím co chcete, ale neobtěžujte mě s tím, že to nefunguje (nejblíž je asi MIT). Jsou to poznatky nasbírané za několik let ne příliš usilovného snažení, takže hodně neúplné, určitě jsou v tom chyby. Nástroj scan-build sice žádnou neodhalil, to však ještě nic neznamená.