Reference

Druhy odkazů v C++

- Reference

`T &, const T &, T &&`

  - Vestavěno v C++
  - Při inicializaci nasměrovány na objekt, **nelze je přesměrovat**
  - Při použití syntakticky identické s hodnotami (r.a)

- (Syrové) ukazatele

`T *, const T *`

  - Vestavěno v C/C++
  - Vyžaduje speciální operátory pro přístup k hodnotě (*p, p->a)
  - **Ukazatelová aritmetika** pro přístup k sousedním hodnotám v polích
  - Ruční dealokace – **používání ve významu vlastníka je dnes nevhodné**

- Chytré ukazatele

`std::shared_ptr< T>, std::unique_ptr< T>`

  - Šablony tříd ve standardní knihovně C++
  - Operátory pro přístup shodné se syrovými ukazateli (*p, p->a)
  - **Reprezentují vlastnictví** – zrušení posledního odkazu vyvolává dealokaci

- Iterátory

`K::iterator, K::const_iterator`

  - Třídy spojené s každým druhem kontejneru (K) ve standardní knihovně C++
  - Vraceny metodami kontejnerů jako **odkazy na prvky kontejnerů**
  - Operátory pro přístup shodné se syrovými ukazateli (*p, p->a)
  - **Ukazatelová aritmetika** pro přístup k sousedním hodnotám v kontejneru

Reference v C++

- Reference lze použít pouze v některých kontextech
  - Formální parametry funkcí (téměř vždy bezpečné a užitečné)
    - Obdoba předávání odkazem v jiných jazycích (ale složitější)
  - Návratové hodnoty funkcí (**nebezpečné** ale někdy nutné)
  - Lokální proměnné (užitečné zvláště jako **auto &&**)
  - Statické proměnné, datové položky tříd (omezené možnosti, vhodnější je ukazatel T * nebo std::reference_wrapper<T>)
- Reference vždy musí být inicializována a nelze ji přesměrovat jinam
  - Všechna použití reference se chovají jako objekt, na který odkazuje
- Reference jsou tří druhů
  - (Modifiable) L-value reference

```
T &
```

    - Skutečný parametr (inicializační výraz) musí být **L-value**, tj. opakovatelně přístupný objekt
  - Const (L-value) reference

```
const T &
```

    - Skutečným argumentem může být libovolný objekt typu T
  - R-value reference

```
T &&
```

    - Skutečný parametr (inicializační výraz) musí být **R-value**, tj. dočasný objekt nebo výraz uzavřený v **std::move()**

Reference v C++

- (Modifiable) L-value reference

`T &`

  - Inicializační výraz musí být modifikovatelná **L-value**, tj. opakovatelně přístupný objekt bez příznaku **const**:
    - **x**, **y.x** - Proměnná nebo datová položka typu **T**, **T&** nebo **T&&**
    - **f()**, **cout<<x** - Volání funkce nebo uživatelsky definovaného operátoru vracející typ **T&**
    - **(T&)e** - výsledek přetypování na **T&**
    - ***p**, **p[i]** - Dereference ukazatele typu **T***, prvek pole typu **T[]** (pro chytré ukazetele či kontejnery jde o uživatelsky definovaný operátor)
    - Několik dalších nevýznamných případů
- Const (L-value) reference

`const T &`

  - Inicializační výraz může být libovolný výraz typu T, tedy L-value i R-value, a navíc nemusí být modifikovatelný
    - Název "const L-value reference" odpovídá syntaxi, ne semantice
    - Výraz formálně nemůže být typu reference, protože reference ve výraze se chová jako odkazovaný objekt. Místo toho se sleduje L-value/R-value kategorie a modifikovatelnost
- R-value reference

`T &&`

  - Inicializační výraz musí být **R-value**:
    - **42**, **'c'**, **true**, **nullptr** - konstanta (vyjma řetězcových)
    - **f(), a+b** - výsledek funkce (nebo operátoru) vracející typ **T** nebo **T&&** (např. **std::move**)
    - **(T)e**, **(T&&)e** - výsledek přetypování na **T** nebo **T&&**
    - Několik dalších nevýznamných případů

Reference v C++

- Proč **T &** vyžaduje L-value
  - T & nejčastěji používán jako "výstupní parametr"
  - Zákaz R-value je ochrana proti chybám programátora

```
void setup(int & p) { p = 42; }

int x;
setup(x);           // OK
setup(x + 1);       // Error
setup(1);           // Error
```

- Proč **const T &** připouští L-value i R-value
  - const T & znamená "vstupní parametr"
  - reference je to jen kvůli rychlosti předávání

```
std::string operator+(const std::string & a, const std::string & b);

std::string x, y, z, u;
u = x + y + z;      // u = ((x + y) + z)
```

  - (x + y) je R-value, přesto ji chceme předat odkazem

- Účel R-value referencí **T &&**
  - Inicializační výraz musí být **R-value**:
    - **42**, **'c'**, **true**, **nullptr** - konstanta (vyjma řetězcových)
    - **f(), a+b** - výsledek funkce (nebo operátoru) vracející typ **T** nebo **T&&** (např. **std::move**)
    - **(T)e**, **(T&&)e** - výsledek přetypování na **T** nebo **T&&**
    - Několik dalších nevýznamných případů
  - R-value se dělí na dva případy
    - *prvalue* - výraz je anonymní dočasný objekt, který nebude víckrát přístupný
    - *xvalue* **-** volání funkce či přetypování vracející **T&&**
- Objekt v R-value výraze lze libovolně modifikovat, protože
  - pro prvalue to nikdo nepozná
    - ale pozor, poběží ještě destruktor objektu
  - pro xvalue s tím programátor souhlasil
    - semantika T&& je "reference na objekt, jehož majitel se o něj již nezajímá"
    - případné další použití objektu původním majitelem musí začít jeho kompletním úklidem
- Nejvýznamnější využití R-value objektu
  - Odebrání a použití dynamicky alokovaných bloků vlastněných objektem
    - Ukazatele uvnitř objektu je nutné vynulovat, aby je jeho destruktor nedealokoval
    - Většinou jako navenek nepřiznaná optimalizace (u kontejnerů ale dokumentovaná)
  - Převzetí vlastnictví bloku drženého chytrým ukazatelem
    - Technicky totéž, ale explicitně publikováno jako součást semantiky chytrých ukazatelů
- Funkce, jejíž parametr je typu T&&, tento parameter pravděpodobně vyprázdní

- **std::move(x)** vrací T&&, tento výraz je tedy R-value
  - uvnitř je přetypování x na T&&
- Funkce, jejíž parametr je typu T&&, tento parameter pravděpodobně vyprázdní
  - Nejčastěji move-constructor a move-assignment
    - oba přemisťují obsah zdroje do cíle
    - move-assignment navíc řeší úklid předchozího obsahu cíle

```
class T {
public:
  T(T && b) noexcept : data_(b.data_) { b.data_ = nullptr; }
  T& operator=(T && b) noexcept {
    T tmp(std::move(b));         // volá move-ctor, tedy vyprázdní b
    std::swap(data_, tmp.data_);// výměna ukazatelů
    return * this;               // následuje volání destruktoru tmp
  }
  ~T() { if (data_) delete data_; }    // destruktor uklízí
private:
  X * data_;    // kdyby zde byl unique_ptr<X>, metody by nebyly zapotřebí
};
```

    - Vlastní implementace move metod je pokročilá technika
    - Většinou se tomu lze vyhnout šikovným použitím chytrých ukazatelů apod.
  - U jiných funkcí se T&& používá v případech nekopírovatelných typů nebo jako pokročilá optimalizace rychlosti

- **std::move(x)** vrací T&&, tento výraz je tedy R-value
  - uvnitř je přetypování x na T&&
  - příklad použití:

```
void swap(T & a, T & b)
{
  T tmp(std::move(a));    // call move-constructor, a is probably emptied
  a = std::move(b);       // call move-assignment, b is probably emptied
  b = std::move(tmp);     // call move-assignment, tmp is no longer needed
}                         // call destructor on tmp
```

    - Skutečné std::swap je šablona - správné řešení je složitější
  - Tři přesuny jsou výrazně rychlejší než tři kopírování
    - Pokud T drží nějaka dynamicky alokovaná data, jejichž vlastnictví lze převzít
    - std::string, std::vector<U> a mnoho dalších, včetně tříd tyto typy obsahujících
  - Řešení s přesuny funguje i pro nekopírovatelné typy
    - std::unique_ptr<X>, std::fstream, std::mutex, ...

- Schopnost levně přesouvat obsahy tříd mění architekturu programů
  - Lze se tak vyhnout dynamické alokaci objektů
  - Např. fstream lze otevřít uvnitř funkce a vrátit jako návratovou hodnotu
    - Příkaz return s lokální proměnnou je automaticky realizován přesunem

Reference v šablonách funkcí a auto

- Univerzální (forwarding) reference
  - Jako argument šablony funkce

```
template< typename T>
void f( T && p)
```

  - Jako proměnná s typem auto

```
auto && x = /*...*/;
```

  - Skutečným argumentem může být R-value i L-value
    - Pozor, v pozadí je *skládání referencí*

```
U a;
auto && x = a;  // type of x is U &
f( a);          // calls f<U &>
```

Argumenty funkcí

Předávání parametrů v C++11

- Pravidla pro typy **formálních argumentů**
  - Pokud funkce potřebuje měnit objekt skutečného argumentu
    - použijte modifikovatelnou referenci: **T &**
  - jinak, pokud je kopírování typu T velmi levné
    - čísla, syrové ukazatele, malé struktury obsahující pouze tyto typy (např. std::complex, iterátory)
    - předávejte hodnotou: **T**
  - jinak, pokud typ T nepodporuje kopírování (např. unique_ptr)
    - předávejte jako R-value reference: **T &&**
  - [pro pokročilé] jinak, pokud funkce někam ukládá kopii parametru
    - pokud opravdu záleží na rychlosti předávání
      - implementujte dvě verze funkce, s **const T &** a **T &&**
    - zjednodušeně
      - předávejte hodnotou: **T**
      - použijte **std::move()** při ukládání
  - jinak
    - použijte konstantní referenci: **const T &**

  - Tato pravidla **neplatí** pro návratové hodnoty a jiné situace

Vstupní parametry předávané odkazem

- Pro vstupní parametry předávané odkazem je nutné použít ***const***
  - Globální funkce:

```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
```

    - Deklarace operátoru + ve standardní knihovně:

```
std::string operator+( const std::string & a, const std::string & b);
```

    - Read-only operátory je vhodné definovat jako globální funkce, jen tak budou fungovat implicitní konverze na levém operandu
  - Metody

```
class my_string {
public:
  my_string concat(const my_string & b) const;
  my_string operator+(const my_string & b) const;
};
```

    - const na konci mění impliticní deklaraci this: const my_string* this

  - Bez *const* není možné použít R-value jako skutečný argument

```
std::string concat(std::string & a, std::string & b);  // WRONG
std::string operator+(std::string & a, std::string & b);  // WRONG

u = concat( concat( x, y), z);      // ERROR
u = x + y + z;      // ERROR
```

Vstupní parametry použité pro inicializaci

- Některé funkce slouží k ukládání hodnot parametrů do trvalejšího úložiště
  - Často jsou to konstruktory

```
class example {
public:
  example( const std::string & a, const std::string & b)
  : a_(a), b_(b) {}
```

  - Dvojtečková sekce konstruktorů slouží ke specifikaci parametrů pro konstruktory jednotlivých součástí (datových položek a předků) třídy
  - Nastavení položek pomocí parametrů konstruktorů je efektivnější než pozdější přiřazení
- Tato varianta konstruktoru je aplikovatelná na L-value i R-value parametry
  - Nedokáže však recyklovat zdroje vlastněné R-value parametry

```
  example( std::string && a, std::string && b)
  : a_(std::move(a)), b_(std::move(b)) {}
```

- Tato varianta konstruktoru recykluje zdroje vlastněné parametry
  - je aplikovatelná pouze na R-value parametry
  - má přednost před const & variantou
- Potřebných kombinací je ale exponenciálně mnoho

```
private:
  std::string a_, b_;
};
```

Vstupní parametry použité pro inicializaci

- Některé funkce slouží k ukládání hodnot parametrů do trvalejšího úložiště

```
class example {
public:
  example( std::string a, std::string b)
  : a_(std::move(a)), b_(std::move(b)) {}
```

  - Pokud je skutečný parametr R-value
    - proměnná a/b bude inicializována move-constructorem
    - položka a_/b_ bude inicializována move-constructorem
  - Pokud je skutečný parametr L-value
    - proměnná a/b bude inicializována copy-constructorem
    - položka a_/b_ bude inicializována move-constructorem
  - Tato varianta řeší exponenciální počet situací najednou
    - Ve srovnání se specializovanými variantami je o jedno volání move-constructoru pomalejší

```
private:
  std::string a_, b_;
};
```

- Předávání hodnotou je vhodné, pokud je recyklace prostředků vždy využita

```
void log( std::string s)  // INEFFICIENT
{
  if (debug_on) log_storage.push_back(std::move(s));
}
```

  - Pokud není debug_on, tato implementace zbytečně kopíruje L-value argument
  - Správné řešení je možné pouze pomocí dvou funkcí s referenčními parametry

Recyklace prostředků pomocí R-value argumentů

- Existují i složitější případy recyklace prostředků
  - Příklad ze standardní knihovny (ve skutečnosti šablona basic_string)

```
string operator+( string && a, const string & b);
```

  - využije alokovaný blok parametru a, pokud je v něm dostatečná rezerva
    - stačí okopírovat obsah b

```
string operator+( const string & a, string && b);
```

  - využije alokovaný blok parametru b, pokud je v něm dostatečná rezerva
    - stačí posunout obsah a, okopírovat obsah b

```
string operator+( string && a, string && b);
```

  - může recyklovat a nebo b, pokud je v některém z nich dostatečná rezerva

```
string operator+( const string & a, const string & b);
```

  - musí alokovat nový prostor a kopírovat oba řetězce
    - stejně jako předchozí verze, pokud nemají dostatečnou rezervu
- Tyto 4 verze nelze nahradit předáváním hodnotou
  - v některých voláních by to způsobilo zbytečné kopírování

- Všechny 4 verze vracejí hodnotou
  - Přestože některé by teoreticky mohly vracet odkaz na svůj R-value parametr
    - To by ale způsobilo problémy v komplikovanějších použitích

```
void f(string & dst, const string & src) { dst = "Hello"; dst += src; }
f(x, std::move(x) + y);
```

    - Programátor předpokládá, proměnná x je po návratu z operátoru + volná
    - Pokud + vrátí odkazem svůj levý parametr (po jeho změně připojením y), dst i src budou reference na proměnnou x

Předávání unique_ptr

```
void send( std::unique_ptr<message> p)
{
  if (channel_closed) throw std::runtime_error("channel_closed");
  channel_buffer.push_back(std::move(p));
}
```

- Tato implementace převezme vlastnictví odkazovaného objektu vždy
  - Pokud je hlášena výjimka, objekt bude dealokován
    - To je v rozporu s "commit or rollback" přístupem:
    - *Pokud funkce vyvolá výjimku, neměla by měnit stav světa*

```
void send( std::unique_ptr<message> && p)
{
  if (channel_closed) throw std::runtime_error("channel_closed");
  channel_buffer.push_back(std::move(p));
}
```

- Tato verze je o jedno volání move-constructoru rychlejší
  - Pokud je hlášena výjimka, ponechá objekt ve vlastnictví volajícího
    - Objekt zanikne později nebo může být volajícím využit:

```
try {
  send(std::move(m));
} catch (...)
{
  send_back(std::move(m));
}
```

- Pro std::shared_ptr by bylo nutné přidat verzi s const & parametrem

Používání chytrých ukazatelů

- Funkce, která nechce převzít (spolu-)vlastnictví:

```
void print( std::unique_ptr<message> p)   // NONSENSE
{
   std::cout << p->data;
}
```

- Předání chytrého ukazatele hodnotou je zde nesmyslné
  - pro std::unique_ptr taková funkce způsobí zánik objektu
    - a vyžaduje std::move ve volání
  - pro std::shared_ptr volání/návrat zbytečně inkrementuje/dekrementuje čítač odkazů

```
void print( const std::unique_ptr<message> & p);
```

- Předání odkazem je funkční, ale zbytečné
  - Tato funkce se nedá použít na nic jiného, než dynamicky alokované objekty

```
void print( const message & m);
```

- Pokud funkce přistupuje k objektu pouze dočasně, nepotřebuje chytrý ukazatel
  - Vyžaduje použití * při volání

```
auto p = std::make_unique<message>("Hello");
print(*p);
```

    - Tato * informuje programátora, že zde nedojde k převzetí vlastnictví objektu
  - Reference umožňuje použití i na jiné, než dynamicky alokované objekty

```
message m("Hello"); print(m);
```

Životnost objektů předávaných odkazem

- Zajištění dostatečné dlouhé životnosti parametrů

```
void print( const message & m);
```

- V běžných případech má objekt předávaný do funkce životnost přesahující dobu běhu funkce

```
message m("Hello"); print(m);
auto p = std::make_unique<message>("Hello"); print(*p);
```

  - Anonymní objekt má zaručenu životnost do konce příkazu:

```
print(message("Hello"));
```

    - vytváří anonymní objekt typu message

```
print(*std::make_unique<message>("Hello"));   // NONSENSE
```

    - neefektivní, ale funkční
    - make_unique vrací hodnotou, výsledek je dočasně uložen v anonymním objektu
    - objekt message je dealokován zánikem chytrého ukazatele na konci příkazu

- Příklady, kdy objekt v parametru stihne zaniknout, jsou krkolomné
  - Předpokládá se, že se jim programátor vyhne

```
void strange(vector<message> & v, const message & m)
{ v.pop_back(); cout << m.data; }
vector<message> k(1, message("Hello"));
strange( k, k.back());
```

  - Často spojeny s race condition

```
message m("Hello"); std::async(print, std::ref(m));
```

    - std::async vyvolá print(m) v jiném vlákně; std::ref vynucuje předání do vlákna odkazem

Životnost objektů předávaných odkazem

- Ukládání odkazů na parametry je inherentně nebezpečné

```
class message_decompressor {
public:
  message_decompressor( const message & m) : m_(m) {}  // DANGEROUS
  std::string decompress_data() const { /*...*/ m_.data /*...*/ }
private:
  const message & m_;
};
```

- Konstruktor ukládá odkaz na svůj parametr
  - Vyžaduje životnost parametru přesahující dobu běhu konstruktoru
  - Při zamýšleném použití to nemusí způsobovat problém:

```
void print( const message & m) {
  message_decompressor d(m);
  std::cout << d.decompress_data();
}
```

  - Chybné použití je nenápadné

```
message_decompressor d(message("Hello"));
std::cout << d.decompress_data();   // CRASH
```

  - Problém skrytý za jinou funkcí

```
message_decompressor get_decompressor(const message &);    // DANGEROUS
```

  - Problém způsobený implicitní konverzí

```
string_decoder get_decoder(const std::string &);           // DANGEROUS
auto d = get_decoder(argv[1]);
std::cout << d.decode_data();    // CRASH
```

    - string_decoder d obsahuje odkaz na dočasný objekt typu std::string vzniklý konverzí

Životnost objektů předávaných odkazem

- Bezpečnější řešení používá parametr typu ukazatel
  - Pokaždé, když funkce ukládá odkaz na parametr někam, kde přežije návrat z funkce

```
class message_decompressor {
public:
  message_decompressor( const message * m) : m_(m) {}
  std::string decompress_data() const { /*...*/ m_->data /*...*/ }
private:
  const message * m_;
};
```

  - Použití ukazatele u datové položky v tomto příkladě nesouvisí s hlavním problémem
    - odstraní vedlejší problém: objekty obsahující reference nelze přiřazovat
- Ukazatel v parametru vynucuje použití operátoru & při volání
  - Toto použití upozorní programátora na potenciální problém

```
void print( const message & m) {
  message_decompressor d(&m);// no problem, d has shorter lifetime than m
  std::cout << d.decompress_data();
}
```

  - Chybné použití způsobí chybové hlášení překladače:

```
message_decompressor d(message("Hello"));
```

    - cannot convert message to const message*

```
message_decompressor d(&message("Hello"));
```

    - operator& requires an L-value

```
string_decoder get_decoder(const std::string *);
auto d = get_decoder(argv[1]);
```

    - cannot convert char* to const std::string*

Používání referencí a ukazatelů jako parametrů

- Funkce, která převezme (spolu-)vlastnictví
  - Výlučné

```
void send( std::unique_ptr<message> && p);
```

  - Sdílené

```
void send( std::shared_ptr<message> p);
```

    - Nebo dvojice funkcí s parametry const & a &&
  - U chytrých ukazatelů lze deklarovat read-only přístup k objektu
    - std::unique_ptr<const X> znamená objekt, který je read-only po celou svou životnost
    - std::shared_ptr<const X> umožňuje read-only spoluvlastníka
- Funkce, která nechce převzít (spolu-)vlastnictví
  - Přistupující k objektu pouze uvnitř funkce:

```
void print( const message & m);
void heal( monster & m);
```

  - Ukládající nevlastnický odkaz *(**observer**)* na objekt na později:

```
message_decompressor::message_decompressor( const message * m);
message_compressor::message_compressor( message * m);
message_decompressor create_decompressor( const message * m);
message_compressor create_compressor( message * m);
```

  - Read-only případy odlišeny použitím const
- Konverze chytrého ukazatele na *
  - Pouze explicitní – upozorňuje programátora na existenci nevlastnických odkazů
  - Metoda .get()
  - Dvojice operátorů &*
    - Funkční i pro iterátory a podobné pseudo-ukazatele, které nemají get

Ukládání nevlastnických odkazů

- Pokud funkce někam ukládá nevlastnický odkaz na objekt, je bezpečnější, když má parametr předávaný jako ukazatel

```
message_compressor create_compressor( message * m);
```

- Řada funkcí standardní knihovny toto pozorování nerespektuje
  - Většinou z dobrých důvodů
  - Programátoři si tyto případy musejí pamatovat a věnovat jejich použití zvláštní pozornost
  - Funkce vracející nějakou formu odkazu na svůj referencí předaný parametr

`std::move, std::forward`

  - Vrací R-value referenci na parametr

`std::ref, std::tie`

  - Vrací hodnotou objekt simulující chování reference na parametr(y)

`std::views::all`

  - Vrací objekt obsahující iterátory do kontejneru, předaného odkazem jako parametr
  - Třídy, obsahující odkazy do objektů předaných jako parametr konstruktoru

`std::string_view`

  - Objekt obsahující ukazatele na vnitřní data objektu std::string, předaného konstruktoru

`std::span`

  - Objekt obsahující ukazatele do pole, předaného (jedné z variant) konstruktoru
  - Metody, vracející odkazy dovnitř objektu, na kterém jsou vyvolány

`.c_str(), .data()`

  - Vrací ukazatel (*) na data uvnitř std::string/std::vector/std::array

`.begin(), .end(), .find(), ...`

  - Vrací iterátor ukazující na element kontejneru
  - begin a end existují i jako globální funkce

Návratové hodnoty funkcí

Předávání návratových hodnot

- Pravidla pro typy **návratových hodnot**
  - Pokud funkce zpřístupňuje objekt v nějaké datové struktuře (např. operator[])
    - a pokud chcete dovolit modifikaci tohoto objektu
      - použijte modifikovatelnou referenci: **T &**
    - jinak
      - použijte konstatní referenci: **const T &**
    - Vracený objekt **MUSÍ PŘEŽÍT** alespoň chvíli **PO NÁVRATU Z FUNKCE**

  - Ve všech ostatních případech
    - předávejte hodnotou: **T**
    - Nepoužívejte **std::move** v příkaze return, pokud v něm je lokální proměnná
    - Překladače mají právo (a někdy povinnost) provést tzv. copy/move-elision C++17
      - ztotožnění vraceného objektu s pomocnou proměnnou připravenou pro návratovou hodnotu v místě volání funkce
      - optimalizace s pozorovatelným efektem na chování programu!

- **POKUD FUNKCE POČÍTÁ** či konstruuje vracenou hodnotu, **NEMŮŽE VRACET ODKAZ**
  - spočtená/zkonstruovaná hodnota je uložena pouze v lokálních objektech, a ty v okamžiku návratu z funkce zaniknou

Vracení odkazem

Funkce vracející odkazem

- Vracení odkazem je možné pouze tehdy, pokud funkce zpřístupňuje data, která budou existovat i po návratu z funkce
  - Všechny následující pokusy jsou nefunkční:
    - anonymní proměnná

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ return Complex(a.re+b.re, a.im+b.im);
  // ERROR: the temporary object will disappear before exiting the function
}
```

    - lokální proměnná

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ Complex tmp(a.re+b.re, a.im+b.im);
  return tmp;  // ERROR: tmp will disappear before exiting the function
}
```

    - globální proměnná

```
Complex tmp;
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ tmp.re = a.re+b.re; tmp.im = a.im+b.im;
  return tmp;  // correct, but unusable for the users
}
Complex v = f(add(x,y), add(z,u));   // which values are passed to f?
```

    - dynamická alokace

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ Complex * tmpp = new Complex(a.re+b.re, a.im+b.im);
  return * tmpp;   // MEMORY LEAK: nobody will deallocate the block
}
```

    - chytrý ukazatel

```
const Complex & add( const Complex & a, const Complex & b) // WRONG
{ std::shared_ptr<Complex> tmpp = std::make_shared<Complex>(a.re+b.re, a.im+b.im);
  return * tmpp;
  // ERROR: the block will be deallocated before exiting the function
}
```

  - Pozor: Všechna tato řešení mohou budit dojem, že fungují

Funkce vracející odkazem

- Vracení odkazem je možné pouze tehdy, pokud funkce zpřístupňuje data, která budou existovat i po návratu z funkce
  - Kvůli *const* objektům jsou obvykle zapotřebí **dvě** funkce
    - Různé hlavičky, většinou (syntakticky) shodná těla
    - Jako globální funkce:

```
T & get_element( std::vector< T> & v, std::size_t i)
{ return v[ i]; }
const T & get_element( const std::vector< T> & v, std::size_t i)
{ return v[ i]; }
```

    - Jako metody:

```
class my_hidden_vector {
public:
  T & get_element( std::size_t i)
  { return v_[ i]; }
  const T & get_element(std::size_t i) const
  { return v_[ i]; }
private:
  std::vector< T> v_;
};
```

    - std::vector má ze stejného důvodu dvě metody operator[]
      - stejná syntaxe v_[i] vede na volání různých funkcí, vracejících odlišné typy

Funkce vracející odkazem

- Nesouměrné varianty jsou obvykle nevhodné
  - Dobrovolné vzdání se práva zápisu:

```
const T & get_element(std::vector< T> & v, std::size_t i) { return v[i]; }
```

    - Tato metoda nevyužívá právo zápisu do v, proto je modifikovatelnost parametru zbytečná
    - Tato situace měla být řešena souměrnou variantou se dvěma const (bez non-const alternativy)

  - Pokus o porušení ochrany (kompilační chyba):

```
T & get_element(const std::vector< T> & v, std::size_t i) { return v[i]; }
```

    - v je const, proto v[i] bude vyřešeno voláním této metody:

```
const T & std::vector<T>::operator[](std::size_t) const;
```

    - Výraz v[i] v příkaze return je tedy typu const T a nesmí být dál předán jako T&

- Logická konstantnost
  - Data vracená operátorem [] jsou fyzicky mimo objekt vector
  - Logicky jsou považována za jeho součást a proto operator[] propaguje const

    - Pravidla samotného jazyka nenutí autory vector<T>::operator[] vracet const T&

```
template< typename T> class vector { public:
  /*const*/ T & operator[](std::size_t i) const { return *(begin_ + i); }
private:
  T * begin_, * end_, * block_end_;
};
```

    - **const** příznak u metody ovlivňuje typ **this** a propaguje se na položky třídy
      - vyjma položek deklarovaných jako **mutable**
      - v const metodě se položky chovají jako by byly deklarovány const:

```
  T * const begin_, * const end_, * const block_end_;
```

    - typ, na který tyto ukazatele ukazují, se tím nemění

Returning by reference

- Overload resolution in C++
  - C++ resolves overloaded function calls using only the types of arguments
    - The use of the result is not considered

```
T & get_element( std::vector< T> & v, std::size_t i);
const T & get_element( const std::vector< T> & v, std::size_t i);
void example1(std::vector<T> & w)
{
  get_element(w,1) = get_element(w,2);   // both calls invoke the first version
}
T example2(const std::vector<T> & w)
{
  return get_element(w,2);  // invokes the second version
}
```

  - With member functions:

```
class my_hidden_vector { public:
  T & get_element( std::size_t i);
  const T & get_element(std::size_t i) const;
};
```

    - The const flag on the object on the left of '.' participates in overload resolution

```
void example1(my_hidden_vector & w)
{
  w.get_element(1) = w.get_element(2);   // both calls invoke the first version
}
T example2(const my_hidden_vector<T> & w)
{
  return w.get_element(2);  // invokes the second version
}
```

Returning by reference

- Interface design in C++
  - Influenced by the life-time considerations for returned values
    - The object must survive the return from the function if returned by reference
  - Returning by value may be slow and exception-unsafe
    - Remedied by C++11/14/17 but still of some concern

```
template< typename T> class vector {
public:
```

    - back() returns the last element which will remain on the stack
      - it may allow modification of the element

```
  T & back();
  const T & back() const;
```

    - pop_back() removes the last element from the stack
      - if it had to return the removed value, it would have to return by value!
      - before C++11: returning by value was slow and exception-unsafe

```
  T pop_back(); // NO SUCH FUNCTION IN std::vector
```

      - therefore, in standard library, the pop_back() function returns nothing

```
  void pop_back();
  // ...
};
```

  - For math-inspired interfaces like operator+, returning by value is necessary
    - Such functions return newly calculated values – no chance to return by reference