Programování v C++

Zápočty a zkoušky

Nov	Domácí úkol #1	0 .. 15 bodů	V každé skupině jiný
Nov .. Dec	Výběr tématu zápočtového programu	Povinné	Dohodou s cvičícím
Dec .. Jan	Domácí úkol #2	0 .. 25 bodů	V každé skupině jiný
Jan .. Feb (Apr)	Praktická zkouška v laboratoři	0 .. 60 bodů	Společně, termíny v SISu
(dle dohody se zkoušejícím)	Nepovinná ústní zkouška	-10 .. +10 bodů	Na žádost studenta
nejpozději May	Odevzdání zápočtového programu	Povinné	Schvaluje cvičící

- Zápočet = 50 bodů + zápočtový program
  - Není nutný pro připuštění ke zkoušce
- Známka = domácí úkoly + praktická zkouška (+ ústní zkouška)
  - 3 = 60 bodů
  - 2 = 75 bodů
  - 1 = 90 bodů

Bodování

Domácí úkol #1	0..15 bodů	-5 bodů za každý započatý týden zpoždění
Domácí úkol #2	0..25 bodů	-10 bodů za každý započatý týden zpoždění
Praktická zkouška	0..60 bodů	50 bodů za plně funkční řešení, +/- 10 bodů za kvalitu zdrojových textů
Nepovinná ústní zkouška	-10..+10 bodů	Podmínkou pro připuštění k ústní části je získání alespoň 50 bodů z předchozích částí hodnocení.

- Zápočet = 50 bodů + zápočtový program
  - Není nutný pro připuštění ke zkoušce
- Známka = domácí úkoly + praktická zkouška (+ ústní zkouška)
  - 3 = 60 bodů
  - 2 = 75 bodů
  - 1 = 90 bodů

Zvláštní případy

- Podmínky mohou být v odůvodněných případech individuálně upraveny: dohodou se cvičícím během října
  - Erasmus students may need dates and deadlines sooner

- Pokud jste předmět nedokončili již loni (nebo dříve)
  - Zapište se ihned do skupiny pro repetenty (NPRG041x05 – Zavoral)
  - Pravidla byla vysvětlena na prvním (a jediném) cvičení

- Pokud předmět nedokončíte letos
  - Vaše body mohou být převedeny do nového roku – přesná pravidla zde: http://www.ksi.mff.cuni.cz/lectures/NPRG041/cviceni/zavoral/repetenti.html
  - Úspěšné odevzdání zápočtového programu bude uznáno, v neúspěšném tématu můžete, ale nemusíte pokračovat

Technical considerations

- Homework assignments
  - Assigned by the teacher of your lab group
  - Submit into recodex.mff.cuni.cz
  - Points assigned by your lab teacher (not by recodex) – visible in SIS
- Practical exam
  - Terms shared by all groups, controlled by randomly assigned teachers
  - Recodex may or may not be used
  - Points assigned by the assigned teacher – visible in SIS
- Programming project
  - Theme agreed between you and your lab teacher (Nov..Dec)
  - Push into gitlab.mff.cuni.cz and inform your lab teacher
  - Repeat until your lab teacher is satisfied

Technical considerations

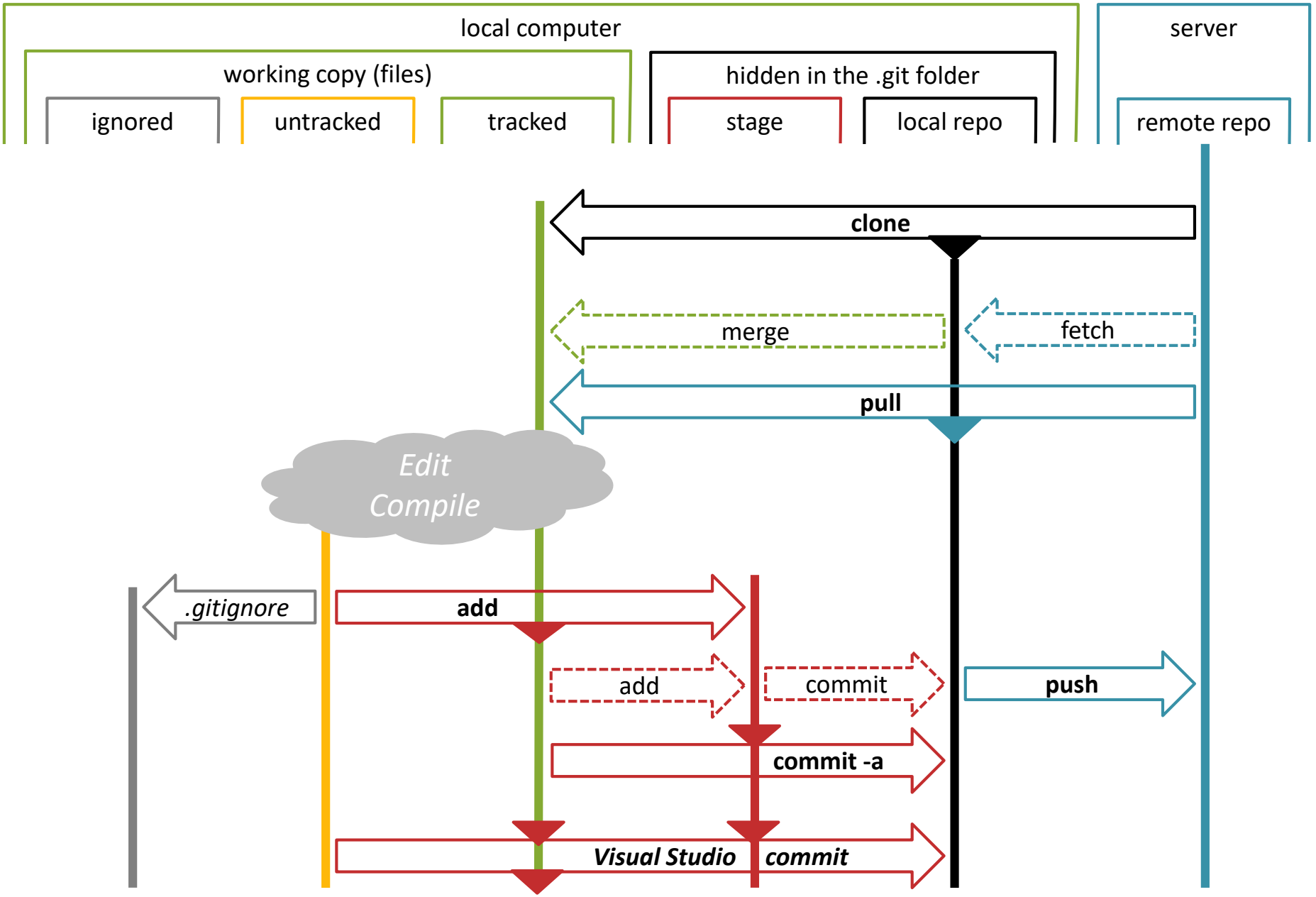
- [today] login into gitlab.mff.cuni.cz
  - Use your SIS credentials (if failed, verify that you can log into ldap1.cuni.cz)
  - This will create your identity in gitlab
- [next week] your lab teacher will create a repository for you
  - You will use it for submitting your programming project
  - You may use it during the development or just push the final version there
- [during October] make sure you can clone the repository
  - Locate the repo at the gitlab web, then copy the "clone" URL to your git client
  - The URL will look like

`https://gitlab.mff.cuni.cz/teaching/nprg041/2019-20/bednarek/vesely.git`

  - You may need to register your public key at the gitlab web

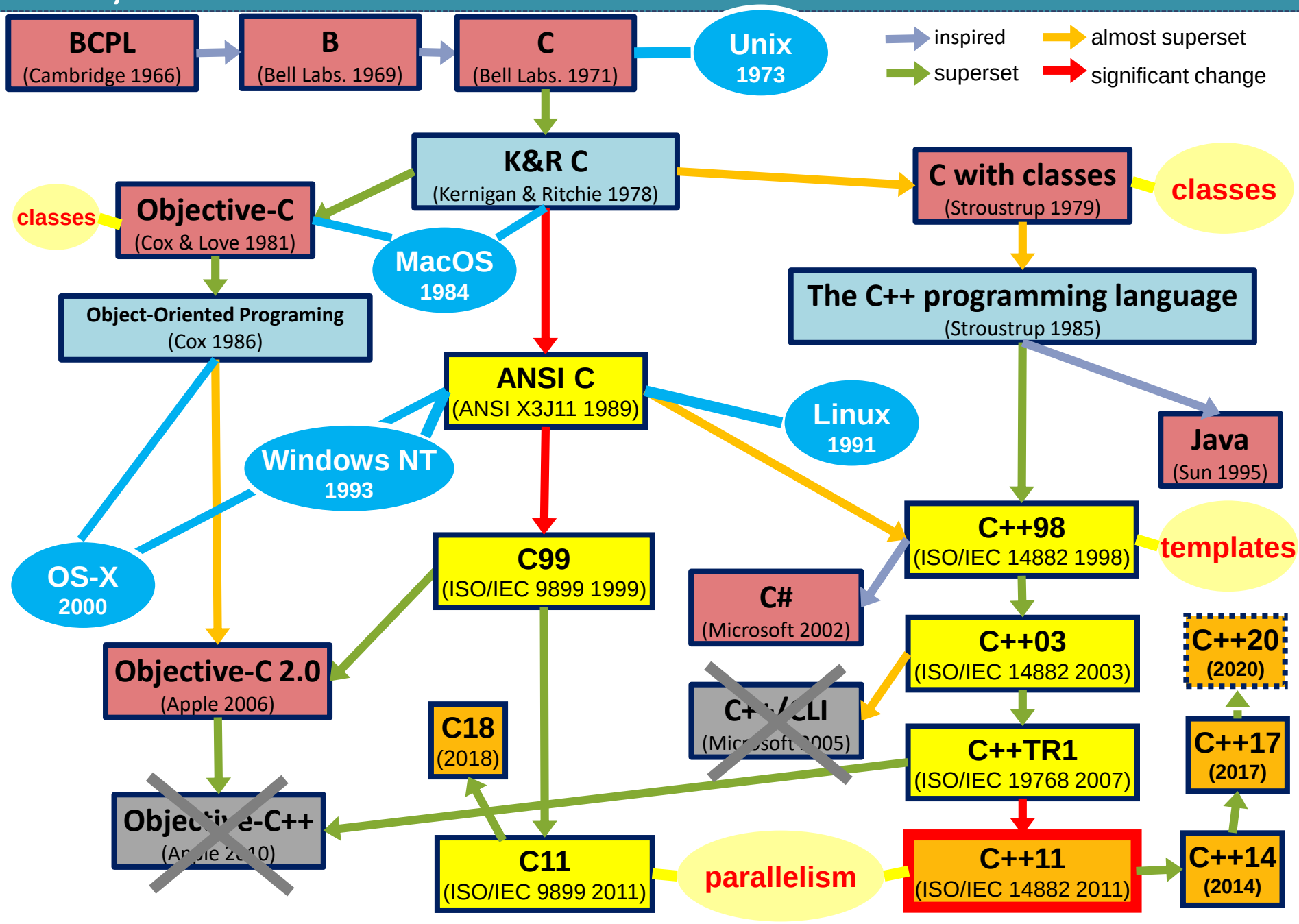
- [soon] Learn the basics of git (www.git-scm.com)
  - Git is the de-facto standard of software industry
    - Built into Microsoft Visual Studio 2019
  - It will help you in any programming project (use github.com or your home server)
  - Your lab teacher may use gitlab to share sample code etc.

Git basics


local computer
working copy (files)
ignored
untracked
tracked
hidden in the .git folder
stage
local repo
server
remote repo
clone
merge
fetch
pull
Edit
Compile
.gitignore
add
add
commit
push
commit -a
Visual Studio
commit

Historie C++

History of C++


BCPL
(Cambridge 1966)
B
(Bell Labs. 1969)
C
(Bell Labs. 1971)
Unix
1973
inspired
almost superset
superset
significant change
K&R C
(Kernigan & Ritchie 1978)
C with classes
(Stroustrup 1979)
classes
classes
Objective-C
(Cox & Love 1981)
MacOS
1984
Object-Oriented Programing
(Cox 1986)
The C++ programming language
(Stroustrup 1985)
ANSI C
(ANSI X3J11 1989)
Linux
1991
Java
(Sun 1995)
Windows NT
1993
C++98
(ISO/IEC 14882 1998)
templates
OS-X
2000
C99
(ISO/IEC 9899 1999)
C#
(Microsoft 2002)
C++03
(ISO/IEC 14882 2003)
C++20
(2020)
Objective-C 2.0
(Apple 2006)
C++/CLI
(Microsoft 2005)
C18
(2018)
C++TR1
(ISO/IEC 19768 2007)
C++17
(2017)
Objective-C++
(Apple 2010)
C11
(ISO/IEC 9899 2011)
parallelism
C++11
(ISO/IEC 14882 2011)
C++14
(2014)

Literatura

Books

- http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Be sure that you have the C++11 versions of the books

- Introduction to programming
  - Stanley B. Lippman, Josée Lajoie, Barbara E. Moo: C++ Primer (5th Edition)
    - Addison-Wesley 2012 (976 pages)
  - Bjarne Stroustrup: Programming: Principles and Practice Using C++ (2nd Edition)
    - Addison-Wesley 2014 (1312 pages)
- Introduction to C++
  - Bjarne Stroustrup: A Tour of C++ (2nd Edition)
    - Addison-Wesley 2018 (256 pages)
- Reference
  - Bjarne Stroustrup: The C++ Programming Language - 4th Edition
    - Addison-Wesley 2013
  - Nicolai M. Josuttis: The C++ Standard Library: A Tutorial and Reference (2nd Edition)
    - Addison-Wesley 2012

Books

- http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Be sure that you have the C++11 versions of the books

- Best practices
  - Scott Meyers: Effective Modern C++
    - O'Reilly 2014 (334 pages)
- Advanced [not in this course]

  - David Vandevoorde, Nicolai M. Josuttis, Douglas Gregor: C++ Templates: The Complete Guide (2nd Edition)
    - Addison-Wesley 2017 (832 pages)
  - Anthony Williams: C++ Concurrency in Action: Practical Multithreading
    - Manning Publications 2012 (528 pages)

- On-line materials
  - Bjarne Stroustrup, Herb Sutter: C++ Core Guidelines
    - github.com/isocpp/CppCoreGuidelines
  - Nate Kohl et al.: C++ reference [C++98, C++03, C++11, C++14, C++17, C++20]
    - cppreference.com

C++


Java Setup - Progress
Java
ORACLE
Status: Installing Java
3 Billion
Devices Run Java
Java
#1 Development Platform
ORACLE

- 3 billion devices run a bytecode interpreter (or JIT compiler), a run-time library, an operating system (if any), and device drivers…

- 3 billion devices run a bytecode interpreter (or JIT compiler), a run-time library, an operating system (if any), and device drivers…

  …implemented mostly in C or C++

- C/C++ can live alone
  - No need for an interpreter or JIT compiler at run-time
  - Run-time support library contains only the parts really required
  - Restricted environments may run with less-than-standard support
    - Dynamic allocation and/or exceptions may be stripped off
    - Code may work with no run-time support at all
  - Compilers allow injection of system/other instructions within C/C++ code
    - Inline assembler or intrinsic functions
  - Code may be mixed with/imported to other languages

- There is no other major language capable of this
  - All current major OS kernels are implemented in C
    - C was designed for this role as part of the second implementation of Unix
    - C++ would be safer but it did not exist
  - Almost all run-time libraries of other languages are implemented in C/C++

- C/C++ is fast
  - Only FORTRAN can currently match C/C++
  - C++ is exactly as fast as C
    - But programming practices in C++ often trade speed for safety
- Why?
  - The effort spent by FORTRAN/C/C++ compiler teams on optimization
    - 40 years of development
  - Strongly typed language with minimum high-level features
    - No garbage-collection, reflexion, introspection, …
  - The language does not enforce any particular programming paradigm
    - C++ is not necessarily object-oriented
  - The programmer controls the placement and lifetime of objects
  - If necessary, the code may be almost as low-level as assembly language
- High-Performance Computing (HPC) is done in FORTRAN and C/C++
- python/R/matlab may also work in HPC well…
  - …but only if most work is done inside library functions (implemented in C)

Major features specific for C++
(compared to other modern languages)

Major distinguishing features of C++ (for beginners)

- Archaic text-based system for publishing module interfaces
  - Will be (gradually) replaced by true modules in C++20
- No 100%-reliable protections
  - Programmer's mistakes may always result in crashes
  - Hard crashes cannot be caught as exceptions
- Preference for value types
  - Similar to old languages (Pascal), unlike any modern language
  - Objects are often manipulated by copying/moving instead of sharing references to them
  - No implicit requirement for dynamic allocation
- No garbage collector
  - Smart pointers provide automatic clean-up since C++11

Major distinguishing features of C++ (for beginners)

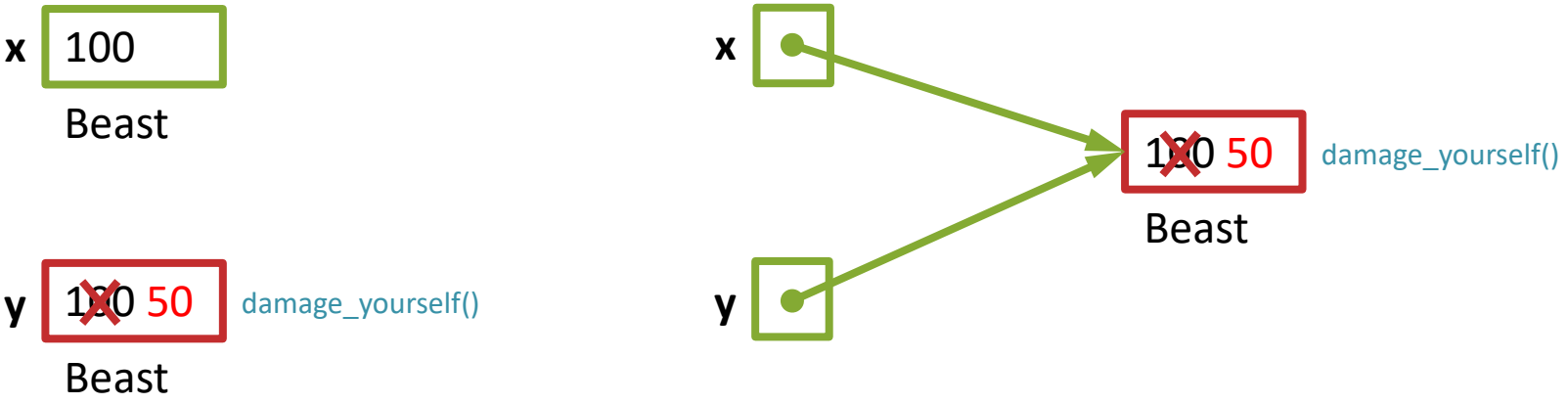
- C makes it easy to shoot yourself in the foot;
  C++ makes it harder, but when you do it blows your whole leg off.
  - Bjarne Stroustrup, creator of C++

Major distinguishing features of C++ (for advanced programmers)

- User-defined operators
  - Pack sophisticated technologies into symbolic interfaces
  - C and the standard library of C++ define widely-used conventions
- No compiler backdoors in the standard library
  - You may create your own library if you don't like the standard
- Extremely strong generic-programming mechanisms
  - Turing-complete compile-time computing environment for meta-programming
  - No run-time component – zero runtime cost of being generic
- C++ is now more complex than any other general programming language ever created

Values vs. references

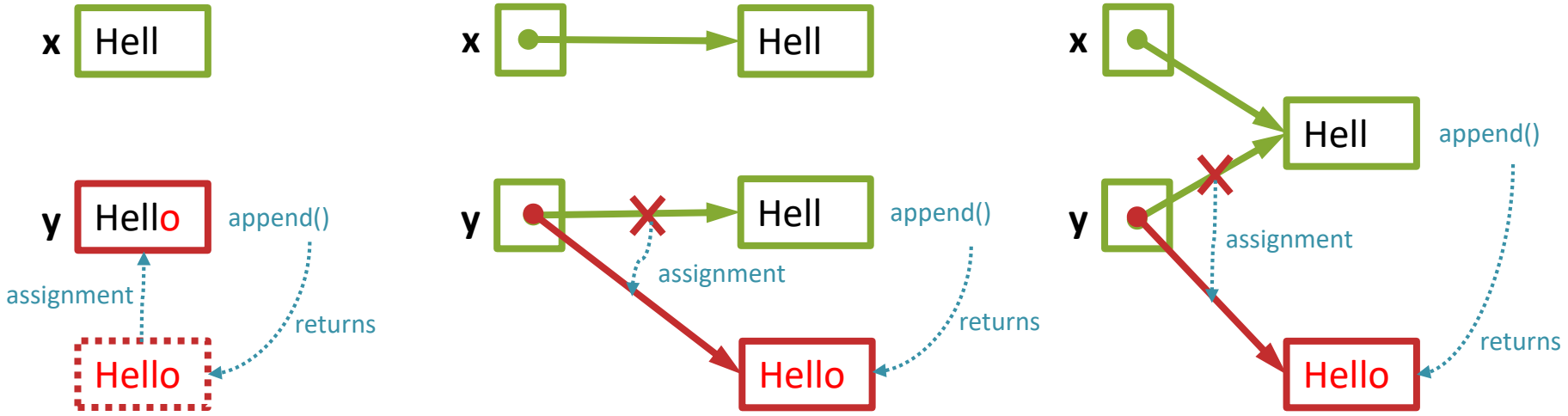
Value vs. reference types



- How does this work in your preferred language?

```
x = create_beast(100);
print(x.health);              // 100
y = x;                        // does it create a copy or shares a reference?
y.damage_yourself(50);
print(x.health);              // 100 if copy, 50 if shared
```

Immutable types



- Note: The distinction is irrelevant for immutable types
  - In many languages (not in C++), strings are immutable

```
x = "Hell";
y = x;                    // is it a copy, deep copy, or shared reference?
// y.append("o");         we cannot tell because we cannot modify y in place
y = y.append("o");        // we only have this interface, returning a new object
```

  - Boxed primitive types (e.g. Integer in java) are usually immutable reference types
  - High-level languages always work with objects – numbers are immutable objects there

```
z = z + 1                 // creates a new object (of type int) in python
```

Value vs. reference types

- How does this work in various languages?

```
x = create_beast(100);
print(x.health);              // 100
y = x;                        // does it create a copy or shared reference?
y.damage_yourself(50);
print(x.health);              // 100 if copy, 50 if shared
```

  - Modern languages are reference-based
    - At least when working with classes and objects
    - Modifying y will also modify x
    - Garbage collector takes care of recycling the memory
  - Archaic languages sometimes give the programmer a choice
    - If x,y are “structures”, assignment copies their contents
      - Records in Pascal, structs in C#, structs/classes in C++
    - If x,y are pointers, assignment produces two pointers to the same object
      - Which pointer is now responsible for deallocating the object?
      - Usually, different syntax is required when accessing members via pointers:

```
x^.health                     (* Pascal *)
(*x).health or x->health      /* C/C++ */
```

Value vs. pointer types in C++

- Variable is the object

```
Beast x, y;
```

  - What are the values now?
    - Defined by the default constructor Beast::Beast()

```
x = create_beast(100);

print(x.health);     // 100
```

  - Assignment copies x over the previous value of y

```
y = x;

y.damage_yourself(50);

print(x.health);     // 100
```

  - Who will kill the Beasts?
    - The compiler takes care

- Variable is a pointer
  - Raw (C) pointers

```
Beast * x, * y;
```

    - Undefined values now!
  - C++11 smart pointers

```
std::shared_ptr< Beast> x, y;
```

  - Different syntax of member access!

```
x = create_beast(100);

print(x->health);    // 100
```

  - Assignment creates a shared reference

```
y = x;

y->damage_yourself(50);

print(x->health);    // 50
```

  - Who will kill the Beast?
    - Raw (C) pointers:

```
delete x; // or y, but not both!
```

    - shared_ptr takes care by counting references (run-time cost!)

Value vs. reference types in C++

- Variable is an object
  - The object may contain a pointer to another object

```
BeastWrapper x, y;


x = create_beast(100);
print(x.health);      // 100
```

  - Assignment does what the author of the class wanted
    - defined by BeastWrapper::operator=

```
y = x;                // ???
y.damage_yourself(50);
print(x.health);      // ???
```

  - C/C++ programmers expect assignment by copy
  - If a class assigns by sharing references, it shall signalize it
    - Name it like “BeastPointer”
    - Use -> for member access (define BeastPointer::operator->)
    - Just like std::shared_ptr
  - However, if the object is immutable or does copy-on-write, it behaves like a value
    - The reference-sharing may remain hidden because it cannot be (easily) detected
  - Who will kill the Beast?
    - The destructor BeastWrapper::~BeastWrapper

Passing by value/reference

Arguments of a function

- Read-only arguments
  - Pass by value
    - For numbers, pointers and small structures/classes (smart pointers, complex)

```
int f( int x) { return x + 1; }
```

  - Pass by **const-reference**
    - For anything large or unknown, including containers and strings (!)

```
std::string g( const std::string & x) { return x + ".txt"; }
```

- Arguments to be modified (including output arguments)
  - Pass by (modifiable) **lvalue-reference**
    - The actual argument must be an **lvalue**

```
void h( int & x) { x = x * 3 + 1; }

void i( std::string & x) { if ( ! x.ends_with(".txt") ) x += ".txt"; }
```

- Arguments to be recycled (advanced trick for low-level libraries)
  - Pass by **rvalue-reference**
    - The actual argument must be an **rvalue** (constant/temporary/std::move(...))

```
std::string j( std::string && x) { x += ".txt"; return x; }
```

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 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;      // decltype(x) == U &
f( a);              // T == U &
```

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)
    - 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

Předávání návratových hodnot v C++11

- 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 C++17
    - Překladače mají právo (a někdy povinnost) provést tzv. copy/move-elision
      - 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

Returning by reference

- Functions which enable access to existing objects may return by reference
  - The object must survive the return from the function

```
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;
```

- this pop_back() removes the last element from the stack and returns its value
- it must return by value - slow (and exception-unsafe, before C++11)

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

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

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

Typické hlavičky funkcí

- Umožnění přístupu k elementu datové struktury

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

- Pokud má celá datová struktura zakázaný zapis, musí být odepřen i pro každý element

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

- Vracení nově zkonstruované hodnoty
  - S lokální proměnnou (C++17: povinné *copy-elision* šetří kopírování)

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

  - S anonymním objektem (povinné *move-elision* šetří kopírování)

```
Complex concat( const Complex & a, const Complex & b)
{ return Complex{ a.re + b.re, a.im + b.im}; }
// nebo: return { a.re + b.re, a.im + b.im};
```

Funkce vracející hodnotou

- Vracení struktur hodnotou

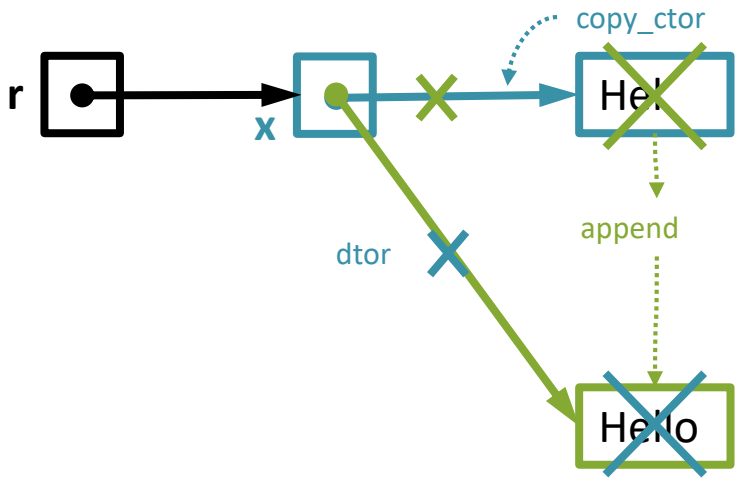
```
std::string concat( const std::string & a, const std::string & b)
{ auto tmp = a; tmp.append( b); return tmp; }
void f()
{ std::string x = concat( y, z); use(x); }
void g()
{ std::string x; x = concat( y, z); use(x); }
```

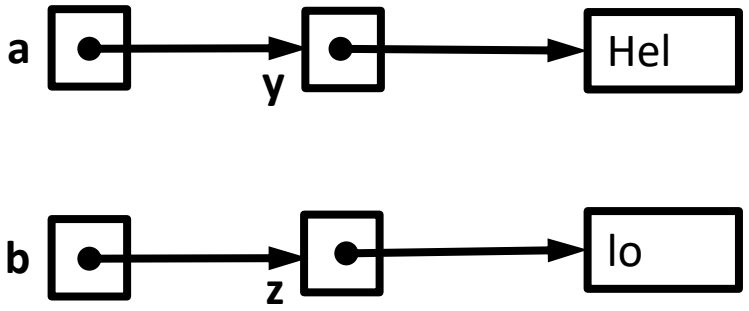
- Překlad s copy-elision (od C++17 v této situaci povinně)
  - Proměnná tmp zaniká, místo ní překladač používá proměnnou ve volající funkci
  - Ekvivalent v hypotetickém jazyce “C s metodami”:

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void f()
{ std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
void g()
{ std::string x,t; x.ctor(); concat(&t,&y,&z); x.move_asgn(&t); t.dtor(); use(x); x.dtor();}
```

  - x.move_asgn(&t) je rychlý *move-assignment*

Funkce vracející hodnotou (v inicializaci)

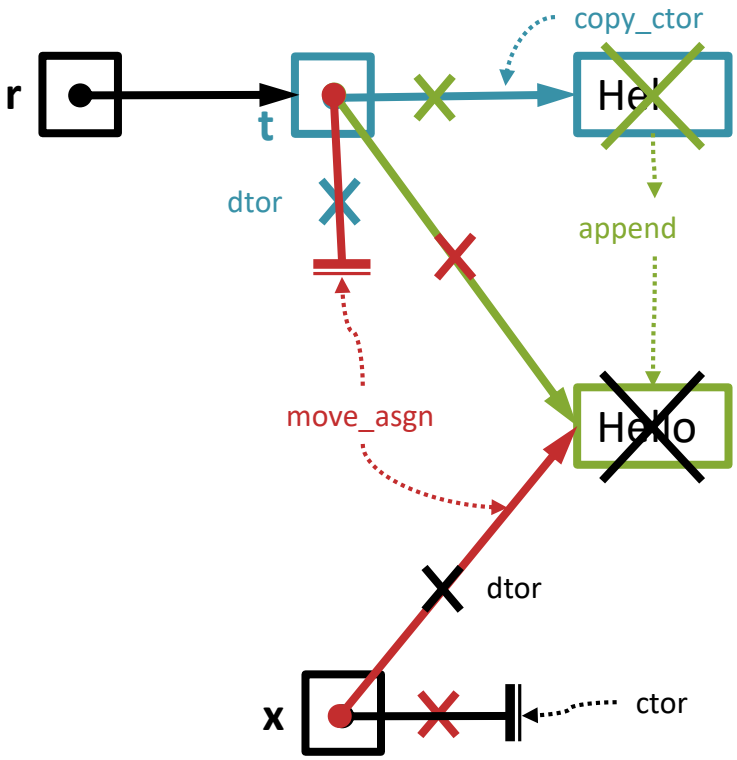


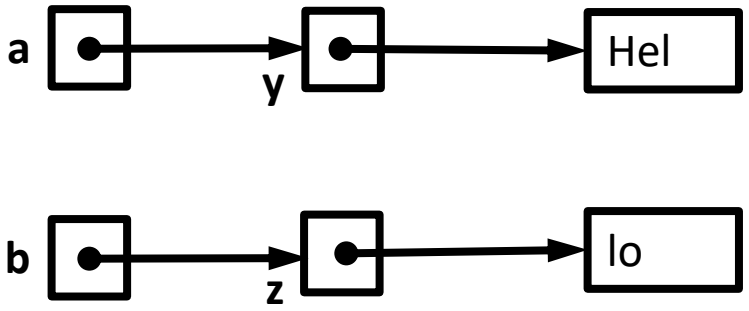


- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void f()
{ std::string x; concat( &x,&y,&z); use(x); x.dtor(); }
```

Funkce vracející hodnotou (v přiřazení)





- C-like equivalent

```
void concat( std::string * r, const std::string * a, const std::string * b)
{ r->copy_ctor(a); r->append(b); }
void g()
{ std::string x,t; x.ctor(); concat(&t,&y,&z); x.move_asgn(&t); t.dtor(); use(x); x.dtor();}
```

Funkce vracející odkazem

- Kvůli *const* objektům jsou 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_;
};
```

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; }
```

  - Metody:

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

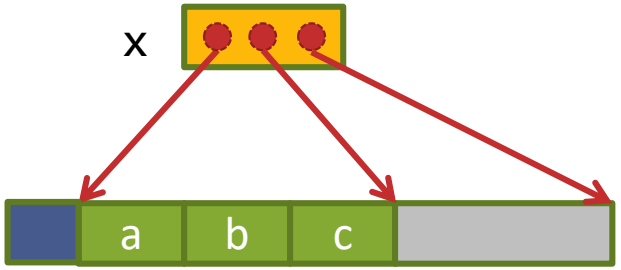
  - Bez *const* není možné použít R-value jako skutečný argument

```
std::string concat2(std::string & a, std::string & b);  // WRONG
u = concat2( concat2( x, y), z);                 // ERROR
```

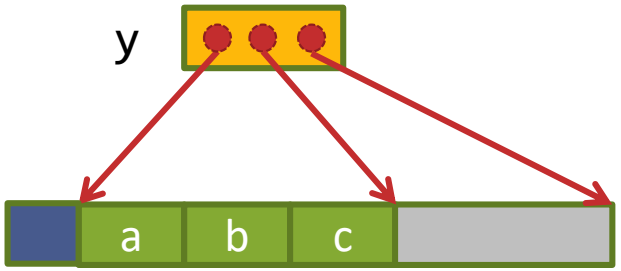
copy/move

Copy

```
std::vector< char> x { 'a', 'b', 'c' };
```



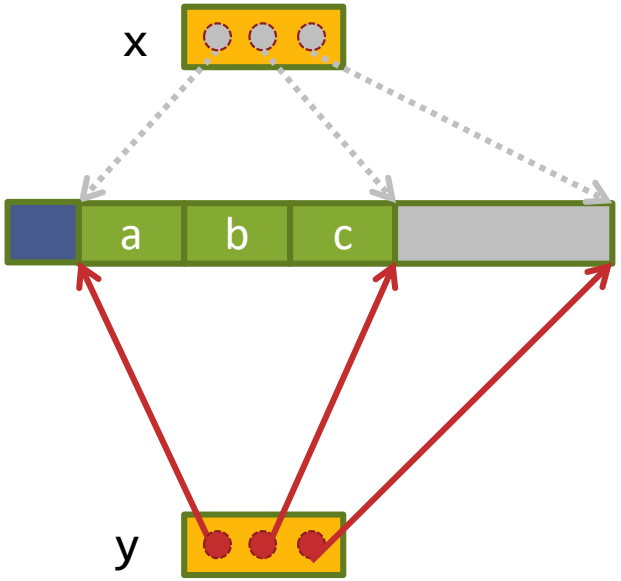

```
std::vector< char> y = x;
```



- Kopírování kontejnerů a podobných objektů
  - Vyžaduje alokaci a kopírování připojených dynamicky alokovaných dat
  - Je pomalé a může vyvolávat výjimky

Move

```
std::vector< char> x { 'a', 'b', 'c' };
```




```
std::vector< char> y = std::move(x);
```

- Po přesunu je zdroj *prázdný*
  - Význam závisí na implementaci konkrétního typu
- Přesun obvykle nepotřebuje alokovat
  - Je rychlý a nevyvolává výjimky

Move

- Přesun (move) je vyvolán namísto kopírování (copy), pokud
  - zdrojový výraz je explicitně označen pomocí std::move(), nebo
  - zdrojový výraz je **r-value**
    - dočasný objekt, který nemůže být použit opakovaně
      - návratové hodnoty funkcí vracejících hodnotou
      - explicitně vytvořené dočasné objekty (T(...), T{...})
      - výsledky přetypování, konstanty apod.

Move

- Semantika *copy* i *move* operací závisí na implementaci typu
  - řeší se čtveřicí speciálních metod
    - copy-constructor – pro inicializaci objektu kopií hodnoty jiného

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

    - move-constructor – pro inicializaci objektu přesunem hodnoty z jiného

```
T( T &&);
```

    - copy-assignment – pro kopírování nové hodnoty namísto staré

```
T & operator=( const T &);
```

    - move-assignment – pro přesun nové hodnoty namísto staré

```
T & operator=( T &&);
```

  - pokud je programátor neimplementuje, překladač je obvykle doplní sám
    - pokud platí určité komplikované podmínky zajišťující zpětnou kompatibilitu
      - v opačném případě nebudou příslušné operace na typu T podporovány
    - překladačem generované implementace aplikují odpovídající speciální metody na všechny složky (a předky)
      - při dodržování doporučených postupů pro programování v C++11 toto chování obvykle vyhovuje
  - pro elementární typy (čísla, T *) je *move* implementováno jako *copy*
    - tím může být narušena konzistence číselných a kontejnerových prvků
  - *move* na kontejneru aplikuje *move* na jednotlivé prvky
    - zdrojový kontejner zůstane prázdný (vyjma std::array)

The Rule of Five

- Consider what happens when your class is going to die...
- ... can all the data members clean-up themselves?
  - Numbers need no clean-up
  - Smart pointers will automatically clean up their memory blocks if necessary
  - Raw (T*) pointers will just disappear, they can not do any clean-up automatically
    - If they are just observers, it is O.K. - they are not responsible for cleaning
    - If they represent ownership, you will need to call delete in a *destructor*

```
class T { public:
  ~T() { delete p_; }          // destructor required
  U * p_;                      // owner of a memory block
};
```

- If you need to write the destructor, you also need to write the four copy/move functions
  - Or to disable them

- Implementing the Five functions is demanding and error-prone
  - Avoid using U* pointers where ownership is required
  - Use only types that can take care of themselves

The Rule of Five – possible scenarios

- All elements support copy and move in the required fashion
  - None of the Five methods required
- All elements support copy and move but copying has no sense
  - Living objects in simulations/games etc.
  - Disable copy methods by "= disable"
  - If move methods remain useful, they should be made accessible by "= default"
- Elements support move in the required fashion, but copying is required
  - Copying elements does not work or behaves differently than required
    - E.g., elements are unique/shared_ptr but the class requires deep copy semantics
  - Implement copy methods, enable move methods by "= default"
- Elements do not support copy/move in the required way
  - Implement all the copy and move methods and the destructor
- Abstract classes must have virtual destructor
  - Required for proper clean-up when objects are deallocated

```
virtual ~C() {}
```

Dynamicky alokovaná data

Dynamická alokace

- Užívejte chytré ukazatele místo syrových (T *)

```
#include <memory>
```

  - Výlučné vlastnictví objektu (ukazatel nelze kopírovat)
    - Zanedbatelné běhové náklady (stejně efektivní jako T *)

```
void f() {
  std::unique_ptr< T> p = std::make_unique< T>();       // alokace T
  std::unique_ptr< T> q = std::move( p);       // přesun do q, p vynulován
}
```

  - Sdílené vlastnictví objektu
    - Netriviální běhové náklady (reference counting)

```
void f() {
  std::shared_ptr< T> p = std::make_shared< T>();       // alokace T
  std::shared_ptr< T> q = p;                    // p a q sdílí objekt
}
```

- Objekt je dealokován v okamžiku zániku posledního vlastníka
  - Destruktor (nebo operator=) chytrého ukazatele vyvolává dealokaci
  - Reference counting nedokáže zrušit cyklické struktury

auto

- Překladač dokáže odvodit typ proměnné z její inicializace
  - std::unique_ptr< T>

```
void f() {
  auto p = std::make_unique< T>();
  auto q = std::move( p);   // pointer moved to q, p becomes nullptr
}
```

  - std::shared_ptr< T>

```
void f() {
  auto p = std::make_shared< T>();   // invokes new
  auto q = p;                // pointer copied to q
}
```

- Pozor na příbuzné typy s automatickými konverzemi
  - k.size() vrací std::size_t který může být větší než int (typ konstanty 0)

```
void f( std::vector< T> & k) {
  for ( auto i = 0; i < k.size(); ++ i)
    /* … k[ i] … */
}
```

Používání ukazatelů v moderním C++

- Vlastník objektu
  - std::unique_ptr< T>, std::shared_ptr< T>
  - Používat pouze tehdy, pokud objekty musejí být alokovány individuálně
    - Možné důvody: Dědičnost, nepravidelná životnost, grafová datová struktura, singleton
    - Pro skladování více objektů stejného typu je výhodnější std::vector< T>
  - std::weak_ptr< T>
    - Rozbití cyklických odkazů při používání std::shared_ptr< T>, používán výjimečně
- Pozorovatel (observer) s právem modifikace
  - T *
    - V moderním C++ syrový ukazatel nemá reprezentovat vlastnictví
  - K uložení odkazu v jiném objektu, který potřebuje **modifikovat** objekt typu T
    - **Pozor na životnost**: Pozorovatel musí přestat pozorovat před zánikem vlastníka
    - Pokud nelze zabránit předčasnému zániku vlastníka, je nutné sdílené vlastnictví
- Read-only observer
  - const T *
  - K uložení odkazu v jiném objektu, který potřebuje **číst** objekt typu T
- Vedle ukazatelů existují i reference (T &, const T &, T &&)
  - Konvence: Reference slouží pro **dočasný** přístup během volání funkce apod.

Owners and observers

- Příklad – unikátní vlastnictví

```
auto owner = std::make_unique< T>();                // std::unique_ptr< T>
```

  - Observer

```
auto modifying_observer = owner.get();   // T *
auto modifying_observer2 = &*owner;                 // same effect as .get()
```

  - Read-only observer

```
const T * read_only_observer = owner.get();         // implicit conversion of T * to const T *
auto read_only_observer2 = (const T *)owner.get(); // explicit conversion
const T * read_only_observer3 = modifying_observer;          // implicit conversion
```

- Ukazatele s vlastnickou semantikou vždy ukazují na celý dynamicky alokovaný blok
- Pozorovatelé smějí ukazovat na jakýkoliv prvek dat
  - Části objektů

```
auto part_observer = & owner->member;
```

  - Statická data

```
static T static_data[ 2];
T * observer_of_static = & static_data[ 0];         // may be redirected to & static_data[ 1]
```

  - Lokální data (pozor na životnost)

```
void g( T * p);
void f() { T local_data; g( & local_data); }
```

Ukazatelé/reference - konvence

Ukazatele vs. reference

- C++ definuje několik druhů odkazů
  - Chytré ukazatele
  - Syrové ukazatele
  - Reference
- Technicky všechny formy umožňují téměř všechno
  - Přinejmenším za použití nedoporučovaných triků
  - Použití referencí je syntakticky odlišné od ukazatelů
- Použití určité formy odkazu signalizuje úmysl programátora
  - Konvence (a pravidla jazyka) omezují možnosti použití předaného odkazu
  - Konvence omezují nejasnosti týkající se extrémnějších situací:
    - Co když někdo zruší objekt, se kterým pracuji?
    - Co když někdo nečekaně modifikuje objekt?
    - …

Předávání ukazatelů a referencí – konvence pro C++11

	Co smí příjemce?	Jak dlouho?	Co mezitím smějí ostatní?
`std::unique_ptr<T>`	Změnit obsah, zrušit objekt	Libovolně	Nic (obvykle)
`std::shared_ptr<T>`	Změnit obsah	Libovolně	Číst/měnit obsah
`T *`	Změnit obsah	Až do dohodnutého okamžiku	Číst/měnit obsah
`const T *`	Číst obsah	Až do dohodnutého okamžiku	Číst/měnit obsah
`T &`	Změnit obsah	Po dobu běhu funkce/příkazu	Nic (obvykle)
`T &&`	Ukrást obsah		Nic
`const T &`	Číst obsah	Po dobu běhu funkce/příkazu	Nic (obvykle)

Ukládání hodnot vedle sebe

Pole a n-tice

	Homogenní (pole)	Polymorfní (n-tice)

Pevná velikost	`// s kontejnerovým rozhraním`
	`static const std::size_t n = 3;`
	`std::array< T, n> a;`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`// struktura/třída`
		`struct S { T1 x; T2 y; T3 z; };`
		`S a;`
		
		`a.x = /*...*/;`
		`a.y.f();`

	`// syrové pole (nedoporučeno)`
	`static const std::size_t n = 3;`
	`T a[ n];`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`// pro generické programování`
		`std::tuple< T1, T2, T3> a;`
		
		`std::get< 0>( a) = /*...*/;`
		`std::get< 1>( a).f();`

Proměnlivá velikost	`std::size_t n = /*...*/;`
	`std::vector< T> a(n);`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`std::vector< std::unique_ptr< Tbase>> a;`
		`a.push_back( std::make_unique< T1>());`
		`a.push_back( std::make_unique< T2>());`
		`a.push_back( std::make_unique< T3>());`
		
		`a[ 1]->f();`

Pole a n-tice v paměti

`std::array< T, 3>`

nebo

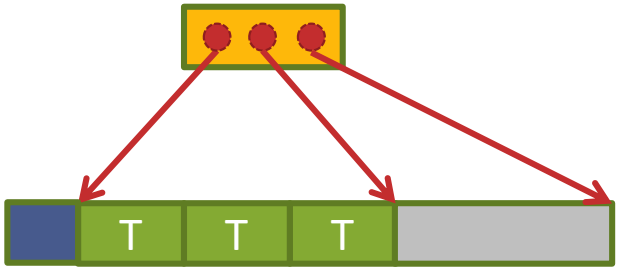
`T[ 3]`

`struct { T1 x; T2 y; T3 z;}`

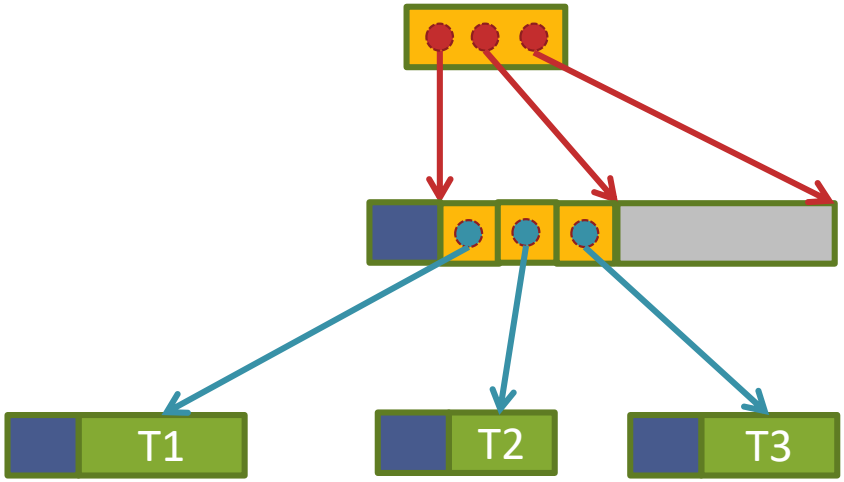
nebo

`std::tuple< T1, T2, T3>`

`std::vector< T>`



`std::vector< std::unique_ptr<Tbase>>`



Smart pointers and containers

<table>
<tr><th></th><th>number of elements</th><th>storage</th><th>ownership</th><th>move</th><th>copy</th></tr>
<tr><td>array<T,N></td><td>fixed N</td><td rowspan="2">inside</td><td rowspan="2">(unique)</td><td rowspan="2">by elements</td><td rowspan="2">by elements</td></tr>
<tr><td>optional<T></td><td rowspan="3">0/1</td></tr>
<tr><td>unique_ptr<T></td><td rowspan="2">individually allocated</td><td>unique</td><td rowspan="7">transfer of ownership</td><td>N.A.</td></tr>
<tr><td>shared_ptr<T></td><td>shared</td><td>sharing</td></tr>
<tr><td>unique_ptr<T[]></td><td rowspan="6">any</td><td rowspan="3">contiguous block</td><td>unique</td><td>N.A.</td></tr>
<tr><td>shared_ptr<T[]></td><td>shared</td><td>sharing</td></tr>
<tr><td>vector<T></td><td rowspan="3">unique</td><td rowspan="3">by elements</td></tr>
<tr><td>deque<T></td><td>several contiguous blocks</td></tr>
<tr><td>other containers</td><td>individually allocated</td></tr>
</table>

Smart pointers and containers

<table>
<tr><th></th><th>number of elements</th><th>storage</th><th>allocation (en masse)</th><th>insert/erase elements</th><th>random access</th></tr>
<tr><td>array<T,N></td><td>fixed, N</td><td rowspan="2">inside</td><td>(when constructed)</td><td>N.A.</td><td>[i]</td></tr>
<tr><td>optional<T></td><td rowspan="3">0/1</td><td>.emplace(...)</td><td rowspan="4">.reset()</td><td rowspan="3">N.A.</td></tr>
<tr><td>unique_ptr<T></td><td rowspan="2">individually allocated</td><td>= make_unique<T>(...)</td></tr>
<tr><td>shared_ptr<T></td><td>= make_shared<T>(...)</td></tr>
<tr><td>unique_ptr<T[]></td><td rowspan="5">any</td><td rowspan="3">contiguous block</td><td>= make_unique<T[]>(n)</td><td rowspan="4">[i]</td></tr>
<tr><td>shared_ptr<T[]></td><td>= make_shared<T[]>(n)</td></tr>
<tr><td>vector<T></td><td rowspan="3">vector<T>(n)
or
.resize(n)</td><td rowspan="2">may move elements</td></tr>
<tr><td>deque<T></td><td>several contiguous blocks</td></tr>
<tr><td>other containers</td><td>individually allocated</td><td>elements never move</td><td>no</td></tr>
</table>

Předávání a sdílení vlastnictví - příklady

Předávání výhradního vlastnictví

```
channel ch;

void send_hello()
{
  auto p = std::make_unique< packet>();
  p->set_contents( "Hello, world!");
  ch.send( std::move( p));
  // p is nullptr now
}

void dump_channel()
{
  while ( ! ch.empty() )
  {
    auto m = ch.receive();
    std::cout << m->get_contents();
    // the packet is deallocated here
  }
}
```

```
class packet { /*...*/ };

class channel
{
public:
  void send( std::unique_ptr< packet> &&);

  bool empty() const;
  std::unique_ptr< packet> receive();

private:
  /*...*/
};
```

Předávání výhradního vlastnictví

```
channel ch;

void send_hello()
{
  auto p = std::make_unique< packet>();
  p->set_contents( "Hello, world!");
  ch.send( std::move( p));
  // p is nullptr now
}

void dump_channel()
{
  while ( ! ch.empty() )
  {
    auto m = ch.receive();
    std::cout << m->get_contents();
    // the packet is deallocated here
  }
}
```

```
class packet { /*...*/ };

class channel
{
public:
  void send( std::unique_ptr< packet> && q)
  {
    q_.push_back( std::move( q));
  }

  std::unique_ptr< packet> receive()
  {
    auto r = std::move( q_.front());
    // remove the nullptr from the queue
    q_.pop_front();
    return r;
  }
private:
  std::deque< std::unique_ptr< packet>> q_;
};
```

Sdílené vlastnictví

```
class sender {
public:
  sender( std::shared_ptr< channel> ch)
    : ch_( std::move( ch)) {}
  void send_hello()
  { /*...*/ ch_->send( /*...*/); }
private:
  std::shared_ptr< channel> ch_;
};

class recipient {
public:
  recipient( std::shared_ptr< channel> ch)
    : ch_( std::move( ch)) {}
  void dump_channel()
  { /*...*/ = ch_->receive(); /*...*/ }
private:
  std::shared_ptr< channel> ch_;
}
```

```
class channel { /*...*/ };

std::unique_ptr< sender> s;
std::unique_ptr< recipient> r;

void init()
{
  auto ch = std::make_shared< channel>();
  s = std::make_unique< sender>( ch);
  r = std::make_unique< recipient>(
                              std::move( ch));
}

void kill_sender()
{ s.reset(); }
void kill_recipient()
{ r.reset(); }
```

- *server* a *recipient* mohou být zrušeny v libovolném pořadí
  - Poslední zruší *channel*

Přístup bez předání vlastnictví

```
class sender {
public:
  sender( channel * ch)
    : ch_( ch) {}
  void send_hello()
  { /*...*/ ch_->send( /*...*/); }
private:
  channel * ch_;
};

class recipient {
public:
  recipient( channel * ch)
    : ch_( ch) {}
  void dump_channel()
  { /*...*/ = ch_->receive(); /*...*/ }
private:
  channel * ch_;
}
```

```
class channel { /*...*/ };

std::unique_ptr< channel> ch;
std::unique_ptr< sender> s;
std::unique_ptr< recipient> r;

void init()
{
  ch = std::make_unique< channel>();
  s = std::make_unique< sender>( ch.get());
  r = std::make_unique< recipient( ch.get());
}

void shutdown()
{ s.reset();
  r.reset();
  ch.reset();
}
```

- *server* a *recipient* musejí být zrušeny před zrušením *channel*
  - Tuto podmínku překladač nedokáže ověřit

Držení ukazatelů na lokální objekty

```
class sender {
public:
  sender( channel * ch)
    : ch_( ch) {}
  void send_hello()
  { /*...*/ ch_->send( /*...*/); }
private:
  channel * ch_;
};

class recipient {
public:
  recipient( channel * ch)
    : ch_( ch) {}
  void dump_channel()
  { /*...*/ = ch_->receive(); /*...*/ }
private:
  channel * ch_;
}
```

```
void do_it( sender &, receiver &);
void do_it_all()
{
  channel ch;
  sender s( & ch);
  recipient r( & ch);

  do_it( s, r);
}
```

- Nutnost použít "&" v parametrech konstruktoru upozorňuje na dlouhý život odkazu
  - "&" - konverze reference na ukazatel
  - "*" – konverze ukazatele na referenci
- Lokální proměnné jsou vždy rušeny v opačném pořadí než jejich konstrukce

Třída obsahující referenci

```
class sender {
public:
  sender( channel & ch)
    : ch_( ch) {}
  void send_hello()
  { /*...*/ ch_.send( /*...*/); }
private:
  channel & ch_;
};

class recipient {
public:
  recipient( channel & ch)
    : ch_( ch) {}
  void dump_channel()
  { /*...*/ = ch_.receive(); /*...*/ }
private:
  channel & ch_;
}
```

```
void do_it( sender &, receiver &);

void do_it_all()
{
  channel ch;
  sender s( ch);
  recipient r( ch);

  do_it( s, r);
}
```

- s a r drží reference na ch po dobu jejich života
  - Bez varování zvláštní syntaxí!
- Jsou-li reference drženy lokálními objekty, vše je v pořádku
- V jiných případech je taková konstrukce nebezpečná

ERROR: Předávání odkazu na lokální objekt mimo jeho životnost

```
class sender {
public:
  sender( channel & ch)
    : ch_( ch) {}
  void send_hello()
  { /*...*/ ch_.send( /*...*/); }
private:
  channel & ch_;
};

class recipient {
public:
  recipient( channel & ch)
    : ch_( ch) {}
  void dump_channel()
  { /*...*/ = ch_.receive(); /*...*/ }
private:
  channel & ch_;
}
```

```
std::unique_ptr< sender> s;
std::unique_ptr< recipient> r;

void init()
{
  channel ch;
  s = std::make_unique< sender>( ch));
  r = std::make_unique< recipient>( ch));
}
```

- ch bude zrušeno dříve než s a r
  - s a r budou přistupovat ke zrušenému objektu
  - Nepředvídatelné katastrofální následky
- Není zde žádné syntaktické varování
  - Proto je v podobných případech vhodnější používat v parametrech konstruktoru ukazatele než reference

ERROR: Dealokace používaného objektu

```
class sender {
public:
  sender( channel & ch)
    : ch_( ch) {}
  void send_hello()
  { /*...*/ ch_.send( /*...*/); }
private:
  channel & ch_;
};

class recipient {
public:
  recipient( channel & ch)
    : ch_( ch) {}
  void dump_channel()
  { /*...*/ = ch_.receive(); /*...*/ }
private:
  channel & ch_;
}
```

```
std::unique_ptr< channel> ch;


void do_it()
{
  ch = std::make_unique< channel>();
  sender s( * ch);
  recipient r( * ch);
  do_it( s, r);
  ch = std::make_unique< channel>();
  do_it( s, r);
}
```

- *ch* je zrušeno dříve než *s* a *r*
  - Dříve či později fatální následky
- Naštěstí je tento styl programování vzácný

STL

Standard Template Library

Kontejnery

- Prefabrikáty základních datových struktur
- Šablony parametrizované typem ukládaného objektu

- Všechny kontejnery pracují s kopiemi vkládaných hodnot
  - Typ hodnot musí mít alespoň copy-constructor a destruktor
  - Některé druhy kontejnerů či operací s nimi vyžadují i operator= nebo konstruktor bez parametrů
- Hodnoty se přidávají a odebírají metodami odpovídajícími druhu kontejneru
- K hodnotám je možno přistupovat pomocí iterátoru, který reprezentuje inteligentní ukazatel dovnitř kontejneru
  - Prostřednictvím iterátoru je možno měnit uložené hodnoty

STL – Příklad

```
#include <deque>

typedef std::deque< int> my_deque;
my_deque the_deque;

the_deque.push_back( 1);
the_deque.push_back( 2);
the_deque.push_back( 3);
int x = the_deque.front(); // 1
the_deque.pop_front();

my_deque::iterator ib  = the_deque.begin();
my_deque::iterator ie  = the_deque.end();
for ( my_deque::iterator it = ib; it != ie; ++it)
{
  *it = *it + 3;
}
int y = the_deque.back(); // 6
the_deque.pop_back()
int z = the_deque.back(); // 5
```

- Sekvenční kontejnery
  - Nové objekty se vkládají na určená místa
    - array< T, N> - pole se staticky danou velikostí
    - vector< T> - pole prvků s přidáváním zprava
      - stack< T> - zásobník
      - priority_queue< T> - prioritní fronta
    - basic_string< T> - vektor s terminátorem
      - string = basic_string< char> - řetězec (ASCII)
      - wstring = basic_string< wchar_t> - řetězec (Unicode)
    - deque< T> - fronta s přidáváním a odebíráním z obou stran
      - queue< T> - fronta (maskovaná deque)
    - forward_list< T> - jednosměrně vázaný seznam
    - list< T> - obousměrně vázaný seznam

STL - Kontejnery

- Asociativní kontejnery
  - Uspořádané (samovyvažující se stromy)
    - set<T> - množina
    - multiset<T> - množina s opakováním
    - map<K,T> - asociativní pole, tj. parciální zobrazení K -> T
    - multimap<K,T> - relace s rychlým vyhledáváním podle klíče K
  - Hashované
    - unordered_set<T> - množina
    - unordered_multiset<T> - množina s opakováním
    - unordered_map<K,T> - asociativní pole, tj. parciální zobrazení K -> T
    - unordered_multimap<K,T> - relace s rychlým vyhledáváním podle klíče K

    - pair<A,B> - pomocná šablona uspořádané dvojice
      - s položkami first, second

STL - Kontejnery

- Uspořádané kontejnery vyžadují uspořádání na klíčích
  - Vystačí si s operací <
  - V nejjednodušším případě to funguje samo

```
std::map< std::string, int> mapa;
```

  - Pokud typ uspořádání nemá, lze jej definovat obecně

```
bool operator<( const Klic & a, const Klic & b) { return ...; }

std::map< Klic, int> mapa;
```

  - Pokud obecná definice uspořádání nevyhovuje, lze definovat uspořádání funktorem pouze pro daný typ kontejneru

```
struct Usporadani {
  bool operator()( const Klic & a, const Klic & b) const { return ...; }
};

std::map< Klic, int, Usporadani> mapa;
```

  - Pokud různé instance kontejneru mají mít různé uspořádání, lze do funktoru doplnit datové položky

```
struct Usporadani { Usporadani( bool a); /*...*/ bool ascending; };

std::map< Klic, int, Usporadani> mapa( Usporadani( true));
```

STL - Kontejnery

- Uspořádání na klíčích – hashující kontejnery
  - Kontejner vyžaduje funktory pro hashování a pro porovnání

```
template< typename K>
class hash { public:
  std::size_t operator()( const K & a) const { /*...*/ }
};
template< typename K>
class equal_to { public:
  bool operator()( const K & a, const K & b) const { return a == b; }
};
```

  - Šablona kontejneru má dva další parametry

```
template< typename K, typename T,
  typename H = std::hash< K>, typename E = std::equal_to< K> >
class unordered_map;
```

  - Konstruktor kontejneru dostává hodnoty funktorů

```
  explicit unordered_map( std::size_t initial_bucket_count = /*...*/,
    const H & h = L(), const E & e = E());
```

STL – Iterátory

- Each container defines two member types: iterator and const_iterator

```
using my_container = std::map< my_key, my_value>;
using my_iterator = my_container::iterator;
using my_const_iterator = my_container::const_iterator;
```

- Iterators act like pointers to objects inside the container
  - objects are accessed using operators *, ->
  - const_iterator does not allow modification of the objects

- An iterator may point
  - to an object inside the container
  - to an imaginary position behind the last object: end()

STL – Iterátory

- Metody kontejneru, vracející iterátory
  - Odkaz na začátek kontejneru - první platný prvek

```
iterator begin()
```

```
const_iterator begin() const
```

  - Odkaz za konec kontejneru - za poslední platný prvek

```
iterator end()
```

```
const_iterator end() const
```

  - iterator a const_iterator jsou typy definované uvnitř kontejneru, zvané iterátory
    - přístupné konstrukcemi jako vector< int>::iterator
    - vlastnosti iterátorů jsou mírně závislé na druhu kontejneru
  - Iterátor kontejneru obsahujícího typ T je třída s operátory definovanými tak, aby se chovala podobně jako "T *" "(ukazatel na typ T) resp. "const T *"
    - Vytváří se tak iluze, že kontejner je pole

STL – Iterátory

```
void example( my_container & c1, const my_container & c2)
{
```

- Každý kontejner definuje metody pro přístup na oba konce; od C++11 též jako globální funkce
  - begin(), cbegin() – první prvek (rovno end() pro prázdný kontejner)
  - end(), cend() – imaginární pozice za posledním prvkem

```
  auto i1 = begin( c1);          // nebo c1.begin(); my_container::iterator
  auto i2 = cbegin( c1);         // nebo c1.cbegin(); my_container::const_iterator
  auto i3 = cbegin( c2);         // nebo c2.cbegin(), begin( c2), c2.begin(); my_container::const_iterator
```

- Asociativní kontejnery umožňují vyhledávání
  - find(k) – první prvek rovný (ani menší, ani větší) k, end() pokud takový není
  - lower_bound(k) – první objekt, který není menší než k, end() pokud takový není
  - upper_bound(k) – první objekt větší než k , end() pokud takový není

```
  my_key k = /*...*/;
  auto i4 = c1.find( k);         // my_container::iterator
  auto i5 = c2.find( k);         // my_container::const_iterator
```

- Iterátory lze posouvat na sousední prvky v kontejneru, případně end()
  - Všechny druhy iterátoru dovolují posun doprava a porovnání na rovnost

```
  for ( auto i6 = c1.begin(); i6 != c1.end(); ++ i6 ) { /*...*/ }
```

  - Obousměrné iterátory (všechny std kontejnery vyjma forward_list) dovolují posun doleva

```
  -- i1;
```

  - Iterátory s náhodným přístupem (vector, string, deque) dovolují přičtení/odečtení čísla, rozdíl a porovnání

```
  auto delta = i4 - c1.begin();  // počet prvků nalevo od i4; my_container::difference_type === std::ptrdiff_t
  auto i7 = c1.end() - delta;    // prvek v dané vzdálenosti zprava; my_container::iterator
  if ( i4 < i7 )
    i4[ delta].f();              // nebo (*(i4 + delta)).f(), (i4 + delta)->f()
}
```

STL – Iterátory

- Pozor:
  - Posouvání iterátoru před begin() nebo za end() je zakázáno

```
for (auto it = c1.end(); it >= c1.begin(); -- it) // ERROR: underruns begin()
```

  - Porovnávání iterátorů mířících do jiných (instancí) kontejnerů je zakázáno

```
if ( c1.begin() < c2.begin() )// ILLEGAL
```

  - Vkládání/odebírání objektů v kontejneru vector/basic_string/deque **invaliduje** všechny iterátory mířící do tohoto kontejneru
    - Vyjma speciálních případů popsaných v detailní dokumentaci
    - Jediný validní iterátor je ten vrácený z volání insert/erase
    - Totéž platí pro ukazatele a reference na data uvnitř kontejneru

```
std::vector< std::string> c( 10, "dummy");

auto it = c.begin() + 5;       // the sixth dummy

std::cout << * it;

auto it2 = c.insert( std::begin(), "first");

std::cout << * it;             // CRASH

it2 += 6;                      // the sixth dummy

c.push_back( "last");

std::cout << * it2; // CRASH
```

    - Obdobné pravidlo platí pro iterátory v unordered_set/map

STL – Insertion/deletion

- Containers may be filled immediately upon construction
  - using n copies of the same object

```
std::vector< std::string> c1( 10, "dummy");
```

  - or by copying from another container

```
std::vector< std::string> c2( c1.begin() + 2, c1.end() - 2);
```

- Expanding containers - insertion
  - insert - copy or move an object into container
  - emplace - construct a new object (with given parameters) inside container
- Sequential containers
  - position specified explicitly by an iterator
    - new object(s) will be inserted before this position

```
c1.insert( c1.begin(), "front");

c1.insert( c1.begin() + 5, "middle");

c1.insert( c1.end(), "back");          // same as c1.push_back( "back");
```

STL – insertion/deletion

- insert by copy
  - slow if copy is expensive

```
std::vector< std::vector< int>> c3;
```

  - not applicable if copy is prohibited

```
std::vector< std::unique_ptr< T>> c4;
```

- insert by move
  - explicitly using std::move

```
auto p = std::make_unique< T>(/*…*/);

c4.push_back( std::move( p));
```

  - implicitly when argument is *rvalue* (temporal object)

```
c3.insert( begin( c3), std::vector< int>( 100, 0));
```

- emplace
  - constructs a new element from given arguments

```
c3.emplace( begin( c3), 100, 0);
```

STL – insertion/deletion

- Shrinking containers - erase/pop
  - single object

```
my_iterator it = /*...*/;

c1.erase( it);

c2.erase( c2.end() - 1);    // same as c2.pop_back();
```

  - range of objects

```
my_iterator it1 = /*...*/, it2 = /*...*/;

c1.erase( it1, it2);

c2.erase( c2.begin(), c2.end());     // same as c2.clear();
```

  - by key (associative containers only)

```
my_key k = /*...*/;

c3.erase( k);
```

Range-for loop

```
for ( type variable : range )
   statement;
```

- range is anything that has begin() and end()
- most often used with universal reference:

```
for ( auto && variable : container )
   statement;
```

- may be used to modify the contents of the *container* by modifying the *variable*

- is by definition equivalent to

```
{
  auto && R = range;
  auto B = begin(R);        // or R.begin() if it exists
  auto E = end(R);          // or R.end() if it exists
  for (; B != E; ++ B)
  { type variable = * B;
    statement;
  }
}
```

Algoritmy

Algoritmy

- Sada generických funkcí pro práci s kontejnery
  - cca 90 funkcí od triviálních po sort, make_heap, set_intersection, ...

```
#include <algorithm>
```

  - Kontejnery se zpřístupňují nepřímo - pomocí iterátorů
    - Obvykle pomocí dvojice iterátorů - polouzavřený interval
    - Lze pracovat s výřezem kontejneru
    - Je možné použít cokoliv, co se chová jako iterátor požadované kategorie
  - Některé algoritmy kontejner pouze čtou
    - Typicky vracejí iterátor
    - Např. hledání v setříděných sekvenčních kontejnerech
  - Většina algoritmů modifikuje objekty v kontejneru
    - Kopírování, přesun (pomocí std::move), výměna (pomocí std::swap)
    - Aplikace uživatelem definované akce (funktor)
  - Iterátory neumožňují přidávání/odebírání objektů v kontejneru
    - Pro "nové" prvky musí být předem připraveno místo
    - Odebrání nepotřebných prvků musí provést uživatel dodatečně

Algoritmy

- Iterátory neumožňují přidávání/odebírání objektů v kontejneru
  - Pro "nové" prvky musí být předem připraveno místo

```
my_container c2( c1.size(), 0);
std::copy( c1.begin(), c1.end(), c2.begin());
```

    - Tento příklad lze zapsat bez algoritmů jako

```
my_container c2( c1.begin(), c1.end());
```

  - Odebrání nepotřebných prvků musí provést uživatel dodatečně

```
auto my_predicate = /*...*/;          // some condition

my_container c2( c1.size(), 0);       // max size
my_iterator it2 = std::copy_if( c1.begin(), c1.end(), c2.begin(), my_predicate);
c2.erase( it2, c2.end());             // shrink to really required size

my_iterator it1 = std::remove_if( c1.begin(), c1.end(), my_predicate);
c1.erase( it1, c1.end());             // really remove unnecessary elements
```

STL – Algoritmy

- Falešné iterátory
  - Algoritmy dovedou pracovat s čímkoliv, co napodobuje iterátor
  - Požadavky algoritmu na iterátory definovány pomocí kategorií
    - Input, Output, Forward, Bidirectional, RandomAccess
  - Každý iterátor musí prozradit svou kategorii a další vlastnosti
    - std::iterator_traits
    - Některé algoritmy se přizpůsobují kategorii svých parametrů (std::distance)
  - Insertery

```
my_container c2;              // empty
auto my_inserter = std::back_inserter( c2);
std::copy_if( c1.begin(), c1.end(), my_inserter, my_predicate);
```

  - Textový vstup/výstup

```
auto my_inserter2 = std::ostream_iterator< int>( std::cout, " ");
std::copy( c1.begin(), c1.end(), my_inserter2);
```

C++20 - ranges

- [C++20] – dvojice iterátorů bude nahrazena jedním ***range***
  - *range* je cokoliv, co má begin() a end()
    - Kontejner je range – tento druh range je vlastníkem dat!
      - kopírování takového range znamená kopírování dat
    - *view* range vznikají jako odkazy na data – nevlastní data
      - view range lze kopírovat v konstantním čase
      - all_view(k) reprezentuje odkaz na všechny prvky kontejneru
      - iota_view(10,20) reprezentuje fiktivní kontejner s obsahem [10,11,...,19]
  - *range adaptor* umožňuje upravit range filtrováním nebo transformací hodnot
    - filter_view(range, pred) vrací range reprezentující pouze prvky splňující predikát
    - adaptéry je možné aplikovat i syntaxí převzatou z unixových rour:

```
range | filter_view(pred)
```

  - Stávající algoritmy dostanou další interface používající *range*
  - *Range* automaticky funguje v [C++11] range-based for

  - *Úplná implementace ani definitivní znění normy zatím (listopad 2019) neexistují*
    - *ranges* jsou knihovna využívající *concepts*; ty jsou velkým rozšířením jazyka [C++20]

Funktory

STL – Funktory

- Příklad - funkce for_each

```
template<class InputIterator, class Function>
Function for_each( InputIterator first, InputIterator last, Function f)
{
    for (; first != last; ++first)
        f( * first);
    return f;
}
```

  - f je cokoliv, co lze zavolat syntaxí f(x)
    - globální funkce (ukazatel na funkci), nebo
    - *funktor*, tj. třída obsahující operator()
  - Funkce f (případně metoda operator()) je zavolána na každý prvek v zadaném intervalu
    - prvek se předává jako * iterator, což může být odkazem
    - funkce f tedy může modifikovat prvky seznamu

STL – Algoritmy

- Jednoduché použití funkce for_each

```
void my_function( double & x)
{
    x += 1;
}
void increment( std::list< double> & c)
{
    std::for_each( c.begin(), c.end(), my_function);
}
```

- [C++11] Lambda
  - Výraz generující funktor (s unikátním anonymním typem)

```
void increment( std::list< double> & c)
{
    for_each( c.begin(), c.end(), []( double & x){ x += 1;});
}
```

STL – Algoritmy

- Předávání parametrů vyžaduje funktor

```
class my_functor {
public:
    double v;
    void operator()( double & x) const { x += v; }
    my_functor( double p) : v( p) {}
};
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), my_functor( value));
}
```

- Lambda s přístupem k lokální proměnné

```
void add( std::list< double> & c, double value)
{
    std::for_each( c.begin(), c.end(), [value]( double & x){ x += value;});
}
```

STL – Algoritmy

- Funktory modifikující svůj obsah

```
class my_functor {
public:
    double s;
    void operator()( const double & x) { s += x; }
    my_functor() : s( 0.0) {}
};
double sum( const std::list< double> & c)
{
    my_functor f = std::for_each( c.begin(), c.end(), my_functor());
    return f.s;
}
```

- Lambda s referencí na lokální proměnnou

```
double sum( const std::list< double> & c)
{   double s = 0.0;
    for_each( c.begin(), c.end(), [& s]( const double & x){ s += x;});
    return s;
}
```

Lambda

Lambda výrazy

- Lambda výraz

[*capture*](*params*) *mutable* -> *rettype* { *body* }

  - Deklaruje třídu ve tvaru

```
class ftor {
public:
  ftor( TList ... plist) : vlist( plist) ... { }
  rettype operator()( params ) const { body }
private:
  TList ... vlist;
};
```

    - vlist je určen proměnnými použitými v *body*
    - TList je určen jejich typy a upraven podle *capture*
    - operator() je const pokud není uvedeno *mutable*
  - Lambda výraz je nahrazen vytvořením objektu

```
ftor( vlist ...)
```

C++11

Lambda výrazy – návratový typ a typ funkce

C++11

- Návratový typ operátoru
  - Explicitně definovaný návratový typ

```
[]() -> int { /*…*/ }
```

  - Automaticky určen pro tělo lambda funkce ve tvaru

```
[]() { return V; }
```

  - Jinak void

C++14

- Generická lambda
  - Parametry typu “auto”, “auto &”, “const auto &”, “auto &&”
    - Nebo dokonce “auto …“ apod. reprezentující libovolný počet argumentů (variadic template)
  - Výsledný operátor je pak šablona funkce

Lambda výrazy – capture

C++11

- Capture
  - Způsob zpřístupnění vnějších entit
    - lokální proměnné
    - this
  - Určuje typy datových položek a konstruktoru funktoru
  - Explicitní capture
    - Programátor vyjmenuje všechny vnější entity v *capture*

```
[a,&b,c,this](){ return a+c+b[c]+m; }
```

    - Entity označené & předány odkazem, ostatní hodnotou
    - **this** umožňuje přístup k položkám objektu (stejně jako v metodě obsahující lambda výraz)
    - Capture s výrazem deklaruje novou proměnnou viditelnou v těle lambda výrazu

```
[x=a+c,&y=b[c],z=m](){ return x+y+z; }
```

  - Implicitní capture (nedoporučuje se)
    - Překladač sám určí vnější entity, *capture* určuje způsob předání

```
[=]
```

```
[=,&b,&y=b[c]]
```

    - předání hodnotou, vyjmenované výjimky odkazem

```
[&]
```

```
[&,a,c]
```

    - předání odkazem, vyjmenované výjimky hodnotou

Class

Class

```
class X {
  /*...*/
};
```

- Třída v C++ je velmi silná univerzální konstrukce
  - Jiné jazyky většinou mívají několik slabších (class+interface)
  - Vyžaduje opatrnost a dodržování konvencí
- Tři stupně použití konstrukce class
  - Ne-instanciovaná třída = balík deklarací (pro generické programování)
  - Třída s datovými položkami a metodami (nejčastější použití v C++)
  - Třída s dědičností a virtuálními funkcemi (objektové programování)
- class = struct
  - struct: prvky implicitně veřejné
    - konvence: neinstanciované třídy a jednoduché třídy s daty
  - class: prvky implicitně privátní
    - konvence: složité třídy s daty a třídy s dědičností

Tři stupně použití class

Non-instantiated class

```
class X {

public:

  using t = int;

  static constexpr int c
  = 1;

  static int f( int p)

  { return p + 1; }

};
```

Class with data members

```
class Y {

public:

  Y() : m_( 0) {}

  int get_m() const

    { return m_; }

  void set_m( int m)

    { m_ = m; }

private:

  int m_;

};
```

Classes with inheritance

```
class U {

public:

  virtual ~U() noexcept {}

  void g() { f_(); }

private:

  virtual void f_() = 0;

};


class V : public U {

public:

  V() : m_( 0) {}

private:

  int m_;

  virtual void f_() override

    { ++ m_; }

};
```

Typy, konstanty a statické položky ve třídách

```
class X {
public:
  class N { /*...*/ };
  typedef unsigned long t;
  using t2 = unsigned long;
  static constexpr t c = 1;
  static t f( t p)
  { return p + v_; }
private:
  static t v_;        // declaration of X::v_
};

X::t X::v_ = X::c;    // definition of X::v_

void f2()
{
  X::t a = 1;
  a = X::f( a);
}
```

- Typové a statické položky...
  - nested class definitions
  - typedef definitions
  - static member constants
  - static member functions
  - static member variables
- ... nejsou vázány na žádnou instanci třídy (objekt)
- Ekvivalentní globálním typům a proměnným, ale
  - Používány s kvalifikovanými jmény (prefix X::)
  - Zapouzdření chrání proti kolizím
    - Ale namespace to dokáže lépe
  - Některé prvky mohou být privátní
  - Třída může být parametrem šablony

Uninstantiated classes vs. namespaces

Uninstantiated class

- Class definitions are intended for objects
  - Static members must be explicitly marked
- Class members may be public/protected/private

```
class X {
public:
  class N { /*...*/ };
  typedef unsigned long t;
  static const t c = 1;
  static t f( t p)
  { return p + v; }
  static t v;           // declaration of X::v
};
```

- Class must be defined in one piece
  - Definitions of class members may be placed outside

```
X::t X::v = X::c;       // definition of X::v

void f2()
{
  X::t a = 1;
  a = X::f( a);
}
```

- A class may become a template argument

```
typedef some_generic_class< X> specific_class;
```

Namespace

- Namespace members are always static
  - No objects can be made from namespaces
  - Functions/variables are not automatically inline/extern

```
namespace X {
  class N { /*...*/ };
  typedef unsigned long t;
  const t c = 1;
  extern t v;           // declaration of X::v
};
```

- Namespace may be reopened
  - Namespace may be split into several header files

```
namespace X {
  inline t f( t p)
  { return p + v; }
  t v = c;              // definition of X::v
};
```

- Namespace members can be made directly visible
  - "using namespace"

```
void f2()
{
  using namespace X;
  t a = 1;
  a = f( a);
}
```

Namespaces and name lookup

```
namespace X {
  class N { /*...*/ };
  typedef unsigned long t;
  const t c = 1;
  extern t v;        // declaration of X::v
  inline t f( N p) { return p.m + v; }
};
```

- Namespace members can be made directly visible
  - "using", "using namespace"
- Functions in namespaces are visible by *argument-dependent lookup*

```
void f2()
{
  X::N a;
  auto b = f( a);    // ADL: calls X::f because the class type of a is a member of X
  using X::t;
  t b = 2;
  using namespace X;
  b = c;
}
```

Class with data members

```
class Y {
public:
  Y()
    : m_( 0)
  {}
  int get_m() const
  { return m_; }
  void set_m( int m)
  { m_ = m; }
private:
  int m_;
};
```

- Class (i.e. type) may be instantiated (into objects)
  - Using a variable of class type

```
Y v1;
```

    - This is NOT a reference!
  - Dynamically allocated
    - Held by a (raw/smart) pointer

```
Y* r = new Y;
std::unique_ptr< Y> p =
  std::make_unique< Y>();
std::shared_ptr< Y> q =
  std::make_shared< Y>();
```

  - Element of a larger type

```
typedef std::array< Y, 5> A;
class C1 { public: Y v; };
class C2 : public Y {};
```

    - Embedded into the larger type
    - NO explicit instantiation by new!

Class with data members

```
class Y {
public:
  Y()
    : m_( 0)
  {}
  int get_m() const
  { return m_; }
  void set_m( int m)
  { m_ = m; }
private:
  int m_;
};
```

- Class (i.e. type) may be instantiated (into objects)

```
Y v1;
std::unique_ptr<Y> p = std::make_unique<Y>();
```

  - Non-static data members constitute the object
  - Non-static member functions are invoked on the object
  - Object must be specified when referring to non-static members

```
v1.get_m()
p->set_m(0)
```

    - References from outside may be prohibited by "private"/"protected"

```
v1.m_      // error
```

    - Only "const" methods may be called on const objects

```
const Y * pp = p.get(); // secondary pointer
pp->set_m(0)         // error
```

Dědičnost

```
class Base { /* ... */ };
class Derived : public Base { /* ... */ }
```

- Třída Derived je odvozena z třídy Base
  - Obsahuje všechny datové položky i metody třídy Base
  - Může k nim doplnit další
    - Není vhodné novými zakrývat staré, vyjma virtuálních
  - Může změnit chování metod, které jsou v Base deklarovány jako virtuální

```
class Base {
  virtual ~Base() noexcept {}
  virtual void f() { /* ... */ }
};

class Derived final : public Base {
  virtual void f() override { /* ... */ }
};
```

- final – zákaz dalších potomků
- override – test existence této virtuální metody v některém předku

Virtuální funkce

```
class Base {
  virtual ~Base() noexcept {}
  virtual void f() { /* ... */ }
};
class Derived : public Base {
  virtual void f() override { /* ... */ }
};
```

- Mechanismus virtuálních funkcí se uplatní pouze v přítomnosti ukazatelů nebo referencí

```
std::unique_ptr<Base> p = std::make_unique< Derived>();      // zde je skryta konverze ukazatelů
p->f();   // volá Derived::f
```

- V jiné situaci není virtuálnost funkcí užitečná

```
Derived d;
d.f();    // volá Derived::f i kdyby nebyla virtuální

Base b = d;          // konverze způsobuje slicing = kopie části objektu
b.f();    // volá Base::f ikdyž je virtuální
```

- Slicing je specifikum jazyka C++

Názvosloví

- Abstraktní třída
  - Definice v C++: Třída obsahující alespoň jednu čistě virtuální funkci
    - Následkem toho nesmí být samostatně instanciována

```
class Base {
  //...
  virtual void f() = 0;
};
```

  - Běžná definice: Třída, která sama nebude instanciována
  - Představuje rozhraní, které mají z ní odvozené třídy (potomci) implementovat

- Konkrétní třída
  - Třída, určená k samostatné instanciaci
  - Implementuje rozhraní, předepsané abstraktní třídou, ze které je odvozena

Dědičnost a destruktor

```
class Base {
public:
  virtual ~Base() noexcept {}
  // ...
};

class Derived : public Base {
public:
  // ...
};
```

```
{
  Base * p = new Derived;
  // ...
  delete p;
}
{
  std::unique_ptr<Base> p =
    std::make_unique< Derived>();
  // ...
  // destruktor unique_ptr volá delete
}
```

- Pokud je objekt destruován operátorem delete aplikovaným na ukazatel na předka, musí být destruktor v tomto předku deklarován jako virtuální

- Odvozené pravidlo:
  - Každá abstraktní třída má mít virtuální destruktor
    - Je to zadarmo
    - Může se to hodit

Dědičnost

- Mechanismus dědičnosti v C++ je velmi silný
  - Bývá používán i pro nevhodné účely
- Ideální použití dědičnosti je pouze toto
  - ISA hierarchie (typicky pro objekty s vlastní identitou)
    - Živočich-Obratlovec-Savec-Pes-Jezevčík
    - Objekt-Viditelný-Editovatelný-Polygon-Čtverec
  - Vztah interface-implementace
    - Readable-InputFile
    - Writable-OutputFile
    - (Readable+Writable)-IOFile

Dědičnost

- ISA hierarchie
  - C++: Jednoduchá nevirtuální veřejná dědičnost

    ```
    class Derived : public Base
    ```

  - Abstraktní třídy někdy obsahují datové položky

- Vztah interface-implementace
  - C++: Násobná virtuální veřejná dědičnost

    ```
    class Derived : virtual public Base1,
                    virtual public Base2
    ```

  - Abstraktní třídy obvykle neobsahují datové položky
  - Interface nebývají využívány k destrukci objektu

  - Oba přístupy se často kombinují

    ```
    class Derived : public Base,
                    virtual public Interface1,
                    virtual public Interface2
    ```

Dědičnost

- Ideální použití dědičnosti je pouze toto
  - ISA hierarchie (typicky pro objekty s vlastní identitou)
  - Vztah interface-implementace
- Jiná použití dědičnosti obvykle signalizují chybu v návrhu
  - Výjimky samozřejmě existují (traits...)
  - Speciálně, dědičnost není správným nástrojem pro reusabilitu kódu
    - Důvod nevhodnosti: automatické konverze

```
void f(container & k) { /* ... */ }

class improved_container : public container { /* some improvement */ };

improved_container ik;

f(ik);    // wrong: f will bypass the improvement (unless everything is virtual)
```

    - Správné řešení je datová položka

```
class improved_container {

  /* proxy methods and some improvement */

  private: constainer m_;

};
```

Nesprávné užití dědičnosti

- Nesprávné užití dědičnosti č. 1

```
class Real { public: double Re; };
class Complex : public Real { public: double Im; };
```

  - Porušuje pravidlo "každý potomek má všechny vlastnosti předka"
    - např. pro vlastnost "má nulovou imaginární složku"
  - Důsledek - slicing:

```
double abs( const Real & p) { return p.Re > 0 ? p.Re : - p.Re; }

Complex x;
double a = abs( x);          // tento kód LZE přeložit, a to je špatně
```

  - Důvod: Referenci na potomka lze přiřadit do reference na předka
    - Complex => Complex & => Real & => const Real &

Nesprávné užití dědičnosti

- Nesprávné užití dědičnosti č. 2

```
class Complex { public: double Re, Im; };

class Real : public Complex { public: Real( double r); };
```

  - Vypadá jako korektní specializace:
    "každé reálné číslo má všechny vlastnosti komplexního čísla"
  - Chyba: Objekty v C++ nejsou hodnoty v matematice
  - Třída Complex má vlastnost "lze do mne přiřadit Complex"
    - Tuto vlastnost třída Real logicky nemá mít, s touto dědičností ji mít bude

```
void set_to_i( Complex & p) { p.Re = 0; p.Im = 1; }


Real x;

set_to_i( x);      // tento kód LZE přeložit, a to je špatně
```

  - Důvod: Referenci na potomka lze přiřadit do reference na předka
  - Real => Real & => Complex &

Speciální metody tříd

Konstruktory a destruktory

- Konstruktor třídy T je metoda se jménem T
  - Typ návratové hodnoty se neurčuje
  - Konstruktorů může být více, liší se parametry
  - Nesmí být virtuální
- Konstruktor je volán vždy, když vzniká objekt typu T
  - Parametry se zadávají při vzniku objektu
  - Některé z konstruktorů mají speciální význam
  - Některé z konstruktorů může generovat sám kompilátor
- Konstruktor nelze vyvolat přímo, pouze pomocí *placement-new*

```
new(p) T(/*...*/);
```

- Destruktor třídy je metoda se jménem ~T
  - Nesmí mít parametry ani návratovou hodnotu
  - Může být virtuální
- Destruktor je volán vždy, když zaniká objekt typu T
  - Destruktor může generovat sám kompilátor
- Destruktor lze vyvolat přímo pouze speciální syntaxí

```
p->T::~T();
```

Speciální metody tříd

- Konstruktor bez parametrů (default constructor)

```
T();
```

  - Používán u proměnných bez inicializace
  - Kompilátor se jej pokusí vygenerovat, je-li to třeba a pokud třída nemá vůbec žádný konstruktor:
    - Položky, které nejsou třídami, nejsou generovaným konstruktorem inicializovány
    - Generovaný konstruktor volá konstruktor bez parametrů na všechny předky a položky
    - To nemusí jít např. pro neexistenci takového konstruktoru

- Destruktor

```
~T() noexcept;
```

  - Používán při zániku objektu
  - Kompilátor se jej pokusí vygenerovat, je-li to třeba a třída jej nemá
  - Pokud je objekt destruován operátorem delete aplikovaným na ukazatel na předka, musí být destruktor v tomto předku deklarován jako virtuální

```
virtual ~T() noexcept;
```

copy/move

- Překladačem definované chování (default)
  - Copy constructor

```
T( const T & x) = default;
```

    - aplikuje copy constructor na složky
  - Move constructor

```
T( T && x) noexcept = default;
```

    - aplikuje move constructor na složky
  - Copy assignment operator

```
T & operator=( const T & x) = default;
```

    - aplikuje copy assignment operator na složky
  - Move assignment operator

```
T & operator=( T && x) noexcept = default;
```

    - aplikuje move assignment operator na složky

  - default umožňuje vynutit defaultní chování

- Podmínky automatického defaultu
  - Copy constructor/assignment operator
    - pokud není explicitně deklarován move constructor ani assignment operator
    - budoucí normy pravděpodobně zakážou automatický default i v případě přítomnosti druhé copy metody nebo destruktoru
  - Move constructor/assignment operator
    - pokud není deklarována žádná ze 4 copy/move metod ani destruktor

Konverze

Speciální metody tříd

- Konverzní konstruktory

```
class T {
  T( U x);
};
```

  - Zobecnění kopírovacího konstruktoru
  - Definuje uživatelskou konverzi typu U na T
  - Je-li tento speciální efekt nežádoucí, lze jej zrušit:

```
explicit T( U v);
```

- Konverzní operátory

```
class T {
  operator U() const;
};
```

  - Definuje uživatelskou konverzi typu T na U
  - Vrací typ U hodnotou (tedy s použitím kopírovacího konstruktoru U, pokud je U třída)

- Kompilátor vždy použije nejvýše jednu uživatelskou konverzi
  - V kombinaci s vestavěnými konverzemi

Přetypování

- Různé varianty syntaxe
  - C-style cast

```
(T)e
```

    - Převzato z C
  - Function-style cast

```
T(e)
```

    - Ekvivalentní (T)e
    - T musí být syntakticky identifikátor nebo klíčové slovo označující typ
  - Type conversion operators
    - Pro odlišení účelu (síly a nebezpečnosti) přetypování:

```
const_cast<T>(e)
```

```
static_cast<T>(e)
```

```
reinterpret_cast<T>(e)
```

    - Novinka - přetypování s běhovou kontrolou:

```
dynamic_cast<T>(e)
```

Přetypování

`const_cast<T>(e)`

- Potlačení konstantnosti
  - const U & => U &
  - const U * => U *

- Obvykle lze nahradit specifikátorem mutable
  - Příklad: Čítač odkazů na logicky konstantní objekt

```
class Data {
public:
  void register_pointer() const
  { references++; }
private:
  /* ... data ... */
  mutable int references;
};
```

Přetypování

`static_cast<T>(e)`

- Umožňuje
- Všechny implicitní konverze
  - Bezztrátové i ztrátové aritmetické konverze (int <=> double apod.)
  - Konverze přidávající modifikátory const a volatile
  - Konverze ukazatele na void *
  - Konverze odkazu na odvozenou třídu na odkaz na předka:
    - Derived & => Base &
    - Derived * => Base *
  - Aplikace copy-constructoru; v kombinaci s implicitní konverzí též:
    - Derived => Base (slicing: okopírování části objektu)
  - Aplikace libovolného konstruktoru T::T s jedním parametrem
    - Uživatelská konverze libovolného typu na třídu T
  - Aplikace konverzního operátoru: operator T()
    - Uživatelská konverze nějaké třídy na libovolný typ T
- ...

Přetypování

`static_cast<T>(e)`

- Umožňuje
- Všechny implicitní konverze
- Konverze na void - zahození hodnoty výrazu
  - Používá se v makrech a podmíněných výrazech
- Konverze odkazu na předka na odkaz na odvozenou třídu
  - Base & => Derived &
  - Base * => Derived *
  - Pokud objekt, na nějž konvertovaný odkaz ukazuje, není typu Derived či z něj odvozený, je výsledek nedefinovaný
    - K chybě obvykle dojde později!
- Konverze celého čísla na výčtový typ
  - Pokud hodnota čísla neodpovídá žádné výčtové konstantě, výsledek je nedefinovaný
- Konverze void * na libovolný ukazatel

Přetypování

```
reinterpret_cast<T>(e)
```

- Umožňuje
  - Konverze ukazatele na dostatečně velké celé číslo
  - Konverze celého čísla na ukazatel
  - Konverze mezi různými ukazateli na funkce
  - Konverze odkazu na odkaz na libovolný jiný typ
    - U * => V *
    - U & => U &
    - Neuvažuje příbuzenské vztahy tříd, neopravuje hodnoty ukazatelů
- Většina použití je závislá na platformě
  - Příklad: Přístup k reálné proměnné po bajtech
  - Typické použití: Čtení a zápis binárních souborů

```
void put_double( std::ostream & o, const double & d)
{ o.write( reinterpret_cast< char *>( & d), sizeof( double)); }
```

  - Obsah souboru je nepřenositelný

Přetypování

`dynamic_cast<T>(e)`

Umožňuje

- Konverze odkazu na odvozenou třídu na odkaz na předka:
  - Derived & => Base &
  - Derived * => Base *
  - Implicitní konverze, chová se stejně jako static_cast
- Konverze odkazu na předka na odkaz na odvozenou třídu
  - Base & => Derived &
  - Base * => Derived *
  - Podmínka: Base musí obsahovat alespoň jednu virtuální funkci
  - Pokud konvertovaný odkaz neodkazuje na objekt typu Derived nebo z něj odvozený, je chování definováno takto:
    - Konverze ukazatelů vrací nulový ukazatel
    - Konverze referencí vyvolává výjimku std::bad_cast
  - Umožňuje přetypování i v případě virtuální dědičnosti

Přetypování

`dynamic_cast<T>(e)`

- Nejčastější použití
  - Konverze odkazu na předka na odkaz na odvozenou třídu

```
class Base { public:
  virtual ~Base() noexcept; /* alespoň jedna virtuální funkce */
};
class X : public Base { /* ... */
};
class Y : public Base { /* ... */
};

Base * p = /* ... */;
X * xp = dynamic_cast< X *>( p);
if ( xp ) { /* ... */ }
Y * yp = dynamic_cast< Y *>( p);
if ( yp ) { /* ... */ }
```

Šablony

Šablony tříd - definice

- Šablona je generická třída parametrizovaná libovolným počtem formálních parametrů těchto druhů:
  - celé číslo – uvnitř šablony se chová jako konstanta, použitelná jako meze polí
  - ukazatel libovolného typu
  - libovolný typ – deklarováno zápisem class T nebo typename T, identifikátor formálního parametru se chová jako identifikátor typu, použitelný uvnitř šablony v libovolné deklaraci
  - šablona třídy s definovanými formálními parametry
  - seznam typů ("variadic template")
- Prefix definice šablony

```
template< formální-parametry>
```

  - lze použít před několika formami deklarací; oblastí platnosti formálních parametrů je celá prefixovaná deklarace

Templates

- Template
  - a generic piece of code
  - parameterized by types and integer constants

- Class templates
  - Global classes
  - Classes nested in other classes, including class templates

```
template< typename T, std::size_t N>
class array { /*...*/ };
```

- Function templates
  - Global functions
  - Member functions, including constructors

```
template< typename T>
inline T max( T x, T y) { /*...*/ }
```

- Type templates [C++11]

```
template< typename T>
using array3 = std::array< T, 3>;
```

Templates

- Template instantiation
  - Using the template with particular type and constant parameters
  - Class and type templates: parameters specified explicitly

```
std::array< int, 10> x;
```

  - Function templates: parameters specified explicitly or implicitly
    - Implicitly - derived by compiler from the types of value arguments

```
int a, b, c;
a = max( b, c);    // calls max< int>
```

    - Explicitly

```
a = max< double>( b, 3,14);
```

    - Mixed: Some (initial) arguments explicitly, the rest implicitly

```
array< int, 5> v;
x = get< 3>( v);   // calls get< 3, array< int, 5>>
```

Templates

- Multiple templates with the same name
  - Class and type templates:
    - one "master" template

```
template< typename T> class vector {/*...*/};
```

    - any number of specializations which override the master template
      - partial specialization

```
template< typename T, std::size_t n> class unique_ptr< T [n]> {/*...*/};
```

      - explicit specialization

```
template<> class vector< bool> {/*...*/};
```

  - Function templates:
    - any number of templates with the same name
    - shared with non-templated functions

Writing templates

- Compiler needs hints from the programmer
  - **Dependent names** have unknown meaning/contents

```
template< typename T> class X
{
```

- type names must be explicitly designated

```
  using U = typename T::B;
  typename U::D p;                    // U is also a dependent name
  using Q = typename Y<T>::C;
  void f() { T::D(); }                // T::D is not a type
```

- explicit template instantiations must be explicitly designated

```
  bool g() { return 0 < T::template h<int>(); }
}
```

- members inherited from dependent classes must be explicitly designated

```
template< typename T> class X : public T
{
  const int K = T::B + 1;             // B is not directly visible although inherited
  void f() { return this->a; }
}
```

Šablony funkcí

- Pod stejným identifikátorem může být deklarováno několik různých šablon funkce a navíc několik obyčejných funkcí.
  - Obyčejné funkce mají přednost před generickými

```
template< class T> T max( T a, T b)
{ return a < b ? b : a; };

char * max( char * a, char * b)
{ return strcmp( a, b) < 0 ? b : a; };

template< int n, class T> T max( Array< n, T> a)
{ /* ... */ }
```

  - Příklad ze standardních knihoven:

```
template< class T> void swap( T & a, T & b)
{ T tmp(a); a = b; b = tmp; };
```

  - K tomu řada chytřejších implementací swap pro některé třídy

Šablony – pokročilé konstrukce jazyka

- Parciální specializace
  - Deklarovanou šablonu lze pro určité kombinace parametrů předefinovat jinak, než určuje její základní definice

```
template< int n> class Array< n, bool>
{ /* specializace pro pole typu bool */ };
```

  - Krajním případem parciální specializace je explicitní specializace
- Explicitní specializace

```
template<> class Array< 32, bool> { /* ... */ };
```

  - U šablon funkcí nahrazena obyčejnou funkcí

- Explicitní instanciace
  - Překladač je možné donutit ke kompletní instanciaci šablony

```
template class Array< 128, char>;
```

Variadic templates

Šablony s proměnlivým počtem parametrů

- Hlavička šablony
  - s proměnlivým počtem typových argumentů

C++11

```
template< typename ... TList>
class C { /* ... */ };
```

- pojmenovaný parametr zastupující seznam typů
- lze i kombinovat s pevnými parametry

```
template< typename T1, int c2, typename ... TList>
class D { /* ... */ };
```

- platí i pro hlavičky parciálních specializací

```
template< typename T1, typename ... TList>
class C< T1, TList ...> { /* ... */ };
```

Šablony s proměnlivým počtem parametrů

C++11

```
template< typename ... TList>
```

- pojmenovaný parametr - seznam typů
- lze uvnitř šablony použít v těchto konstrukcích:
  - vždy se suffixem ...
  - typové argumenty v použití (jiné) šablony

```
X< TList ...>
Y< int, TList ..., double>
```

  - seznam předků třídy

```
class E : public TList ...
```

  - deklarace parametrů funkce/metody/konstruktoru

```
void f( TList ... plist);
double g( int a, double c, TList ... b);
```

  - tím vzniká pojmenovaný parametr zastupující seznam hodnot
  - ke každému seznamu hodnot musí být seznam typů

  - několik dalších okrajových případů
  - počet prvků seznamu

```
sizeof...(TList)
```

Šablony s proměnlivým počtem parametrů

C++11

```
template< typename ... TList>
  void f( TList ... plist);
```

- **pojmenovaný parametr - seznam hodnot**
- lze uvnitř funkce použít v těchto konstrukcích:
  - vždy se suffixem ...
  - hodnotové argumenty ve volání (jiné) funkce/konstruktoru

```
g( plist ...)
new T( a, plist ..., 7)
T v( b, plist ..., 8);
```

- inicializační sekce konstruktoru

```
E( TList ... plist)
: TList( plist) ...
{
}
```

- několik dalších případů

Šablony s proměnlivým počtem parametrů

C++11

```
template< typename ... TList>
  void f( TList ... plist);
```

- **při použití je možno prvky seznamu obalit**
  - suffix ... slouží jako kompilační for_each
  - (každý) výskyt názvu seznamu je nahrazen jeho i-tým prvkem
- výsledkem je
  - seznam typů v parametrech šablony nebo deklaraci funkce

```
X< std::pair< int, TList *> ...>
class E : public U< TList> ...
void f( const TList & ... plist);
```

- seznam výrazů ve volání funkce/metody/konstruktoru

```
g( make_pair( 1, & plist) ...);
h( static_cast< TList *>( plist) ...);
i( sizeof( TList) ...); // pozor, sizeof...( TList) je něco jiného
```

- seznam inicializátorů v konstruktoru
- další okrajové případy

Generická N-tice

```
template <class ... Types> class tuple {
public:
  tuple( const Types & ...);
  /* black magic */
};
template < size_t I, class T>
  class tuple_element {
public:
  typedef /* black magic */ type;
};
template < size_t I, class ... Types>
  typename tuple_element< I, tuple< Types ...> >::type &
  get( tuple< Types ...> & t);
```

C++11: <utility>

- použití

```
typedef tuple< int, double, int> my_tuple;
typedef typename tuple_element< 1, my_tuple>::type alias_to_double;

my_tuple t1( 1, 2.3, 4);
double v = get< 1>( t1);
```

Programming languages and compilers

Human-readable and machine-readable code

- Human-readable code (C/C++)

```
a = b - c;
```

- Human-readable assembly code (Intel/AMD x86)

```
mov eax,dword ptr [b]
sub eax,dword ptr [c]
mov dword ptr [a],eax
```

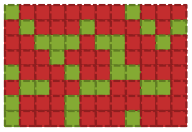
- Less readable assembly code

```
mov eax,dword ptr [rsp+30h]
sub eax,dword ptr [rsp+38h]
mov dword ptr [rsp+40h],eax
```

- Human-readable binary code

```
8B 44 24 30
2B 44 24 38
89 44 24 40
```

- "Machine readable" binary code

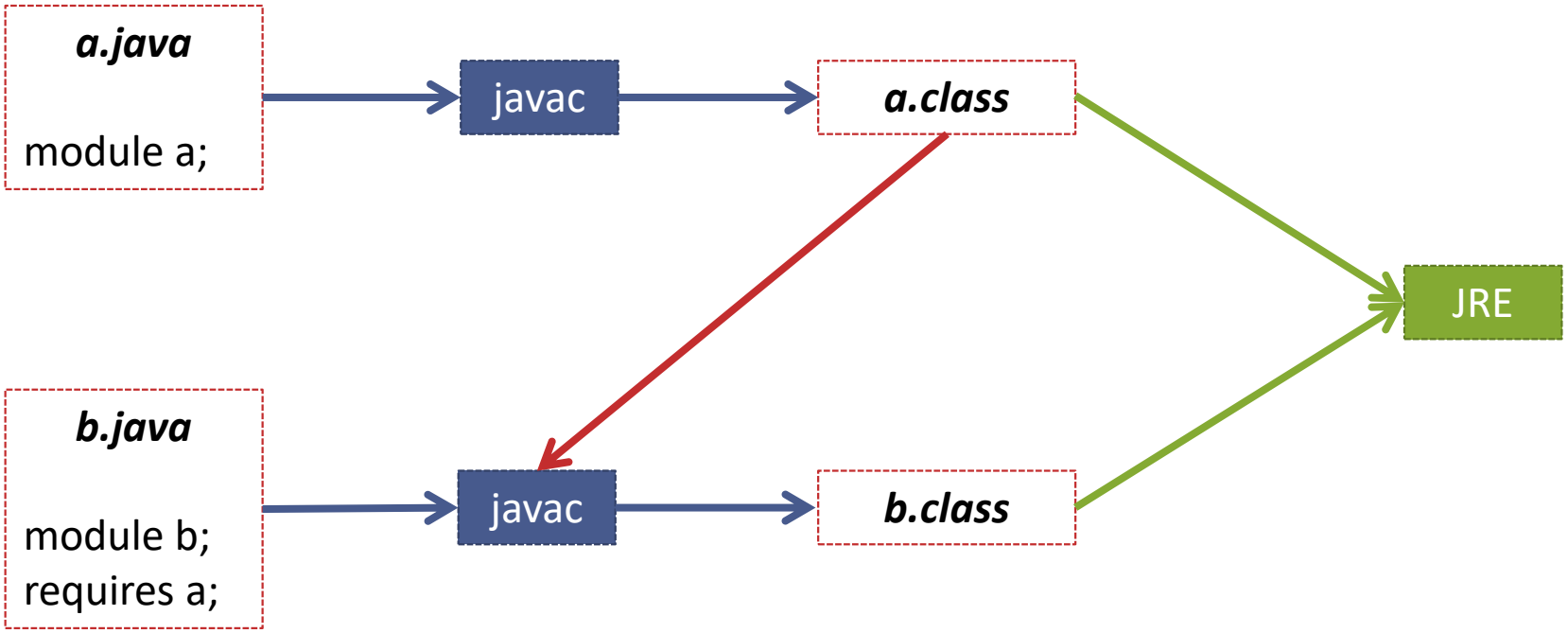


- The role of the compiler
  - Allocate space for the variables

```
a=[rsp+40h], b=[rsp+30h], c=[rsp+38h]
```

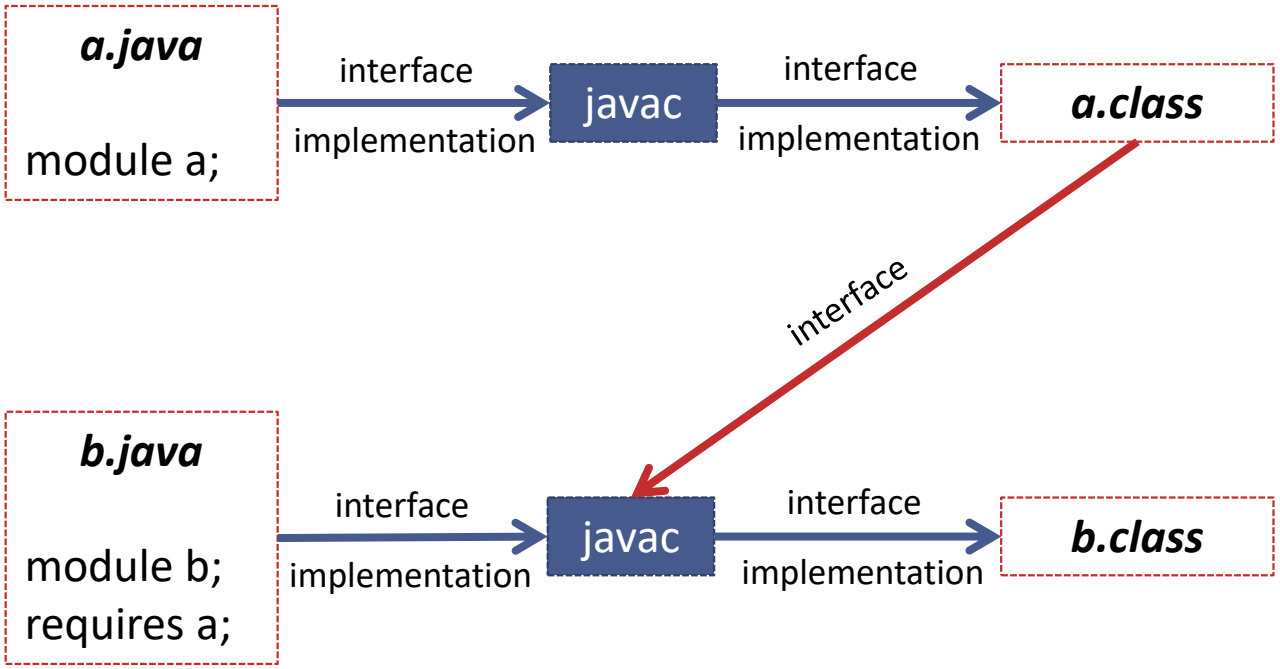
  - Find the declarations of the names
  - Determine their types (32-bit int)
  - Check applicability of operators
- In C/C++, the result of compilation is binary code of the target hardware
  - Find corresponding instruction(s)
    - "sub" – integer subtraction
    - "dword ptr" - 32-bit
  - Allocate registers for temporaries
    - "eax" - a 32-bit register
  - … and many other details
- Produce a stand-alone executable
  - Loadable by the operating system

Compilation in modern languages



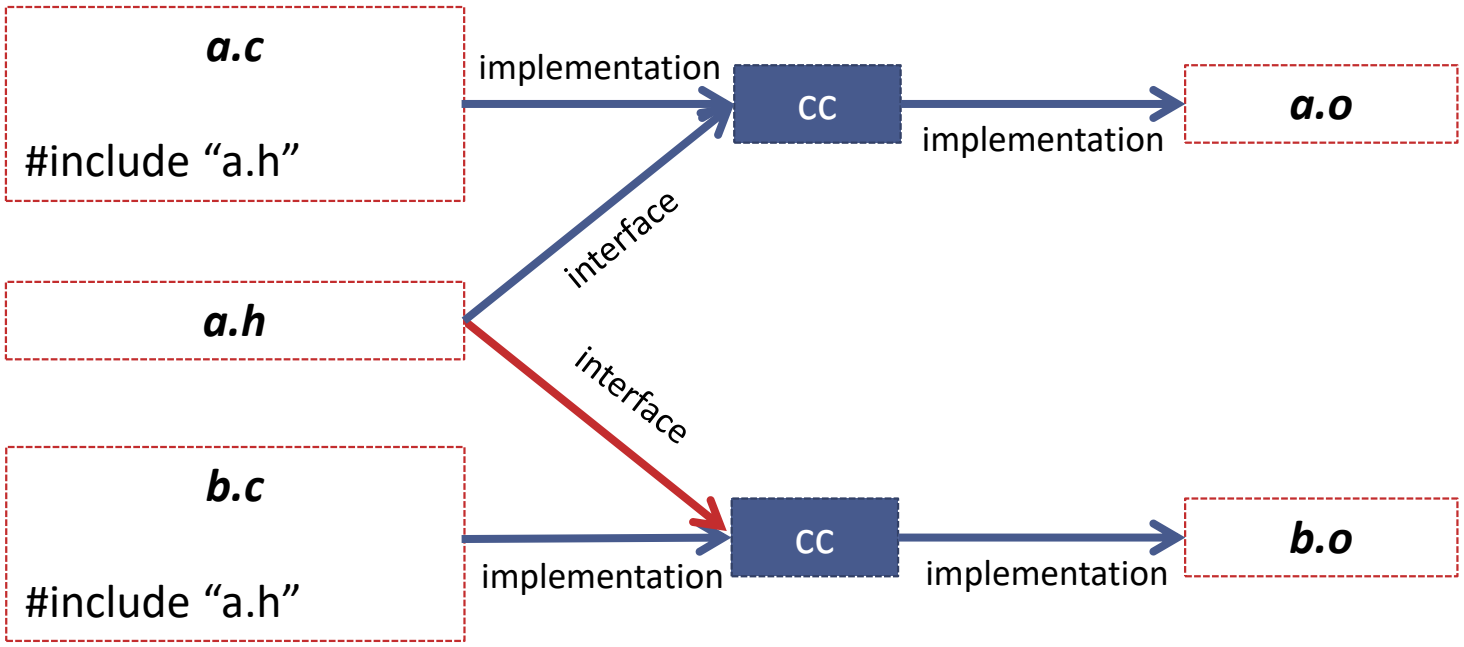
- Most modern languages compile source code into binary packages
  - These packages are also read by the compiler when referenced
- But not in C/C++ (yet)
  - Discussed for years, no conclusion reached yet – in C++20 or later

Compilation in modern languages



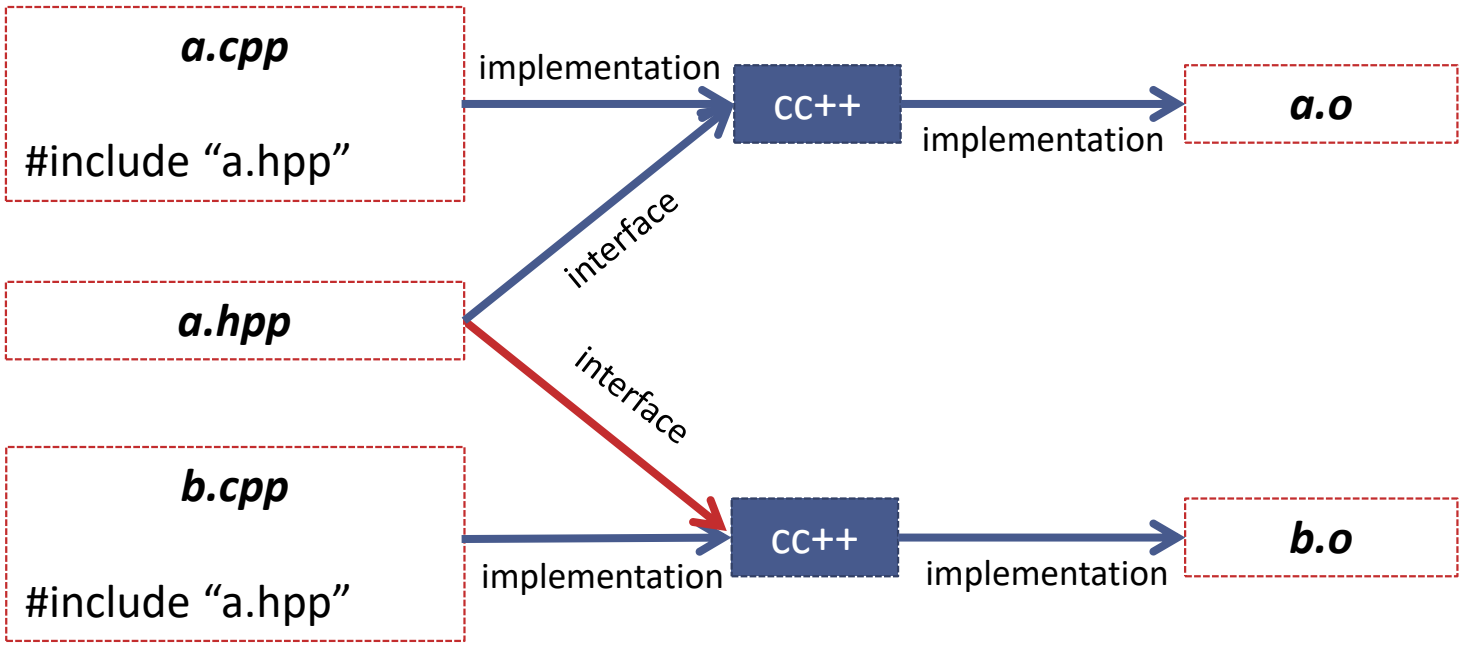
- Why not in C/C++? There are disadvantages:
  - When anything inside a.java changes, new timestamp of a.class induces recompilation of b.java
    - Even if the change is not in the public interface
  - How do you handle cyclic references?

Compilation in C



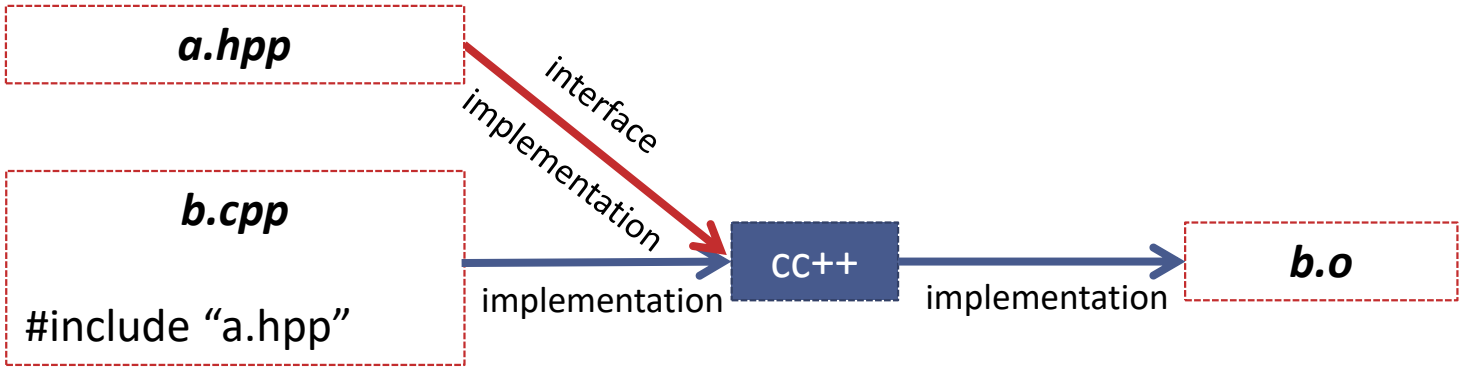
- In C, the situation was simple
  - Interface = function headers in „header files“
    - Typically small
  - Implementation = function bodies in “C files”
    - Change of a.c does not require recompilation of b.c

Compilation in C++



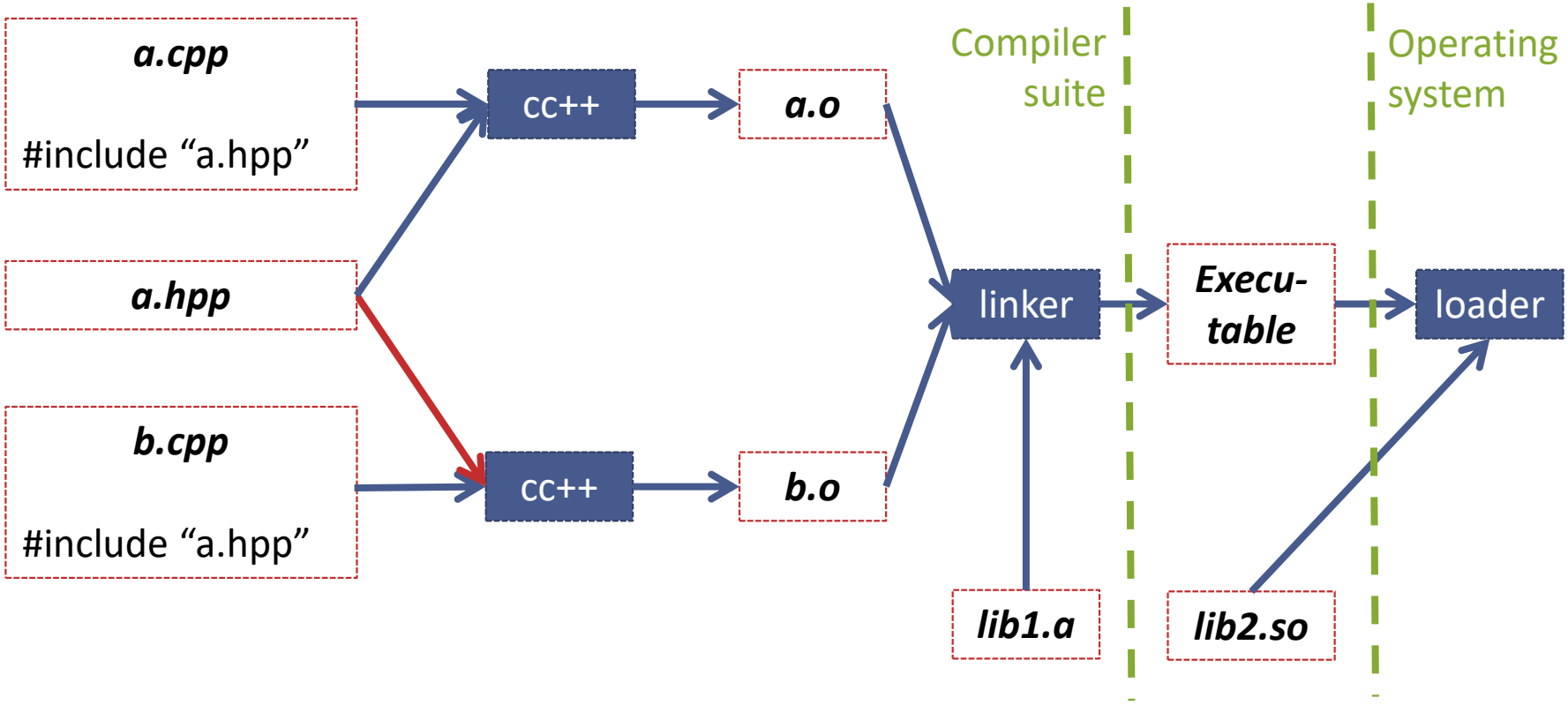
- In modern C++, the separate compilation is no longer advantage
  - Interface (classes etc.) is often larger than implementation (function bodies)
  - Changes often affect the interface, not the body
- The purely textual behavior of #include is anachronism

Compilation of generic code in C++



- Implementation of generic functions (templates) must be visible where called
  - Explanation later…
- Generic code often comprises of header files only

Compilation in C++



- Object files (.o, .obj) contain binary code of target platform
  - They are incomplete – not executable yet
- Linker/loader merges them together with library code
  - Static/dynamic libraries. Details later…

Hello, World!

Hello, World!

```
#include <iostream>


int main( int argc, char * * argv)
{
  std::cout
    << "Hello, world!"
    << std::endl;

  return 0;
}
```

- Program entry point
  - Heritage of the C language
    - No classes or namespaces
  - Global function "main"
- main function arguments
  - Command-line arguments
    - Split to pieces
  - Archaic data types
    - Pointer to pointer to char
    - Logically: array of strings
- std - standard library namespace
- cout - standard output
  - global variable
- << - stream output
  - overloaded operator
- endl - line delimiter
  - global function (trick!)

Hello, World!

More than one module

- Module interface described in a file
  - .hpp - "header" file
- The defining and all the using modules shall "include" the file
  - Text-based inclusion

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

void world();

#endif
```

```
// main.cpp
#include "world.hpp"


int main( int argc, char * * argv)
{
  world();
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>


void world()
{
  std::cout << "Hello, world!"
      << std::endl;
}
```

Hello, World!

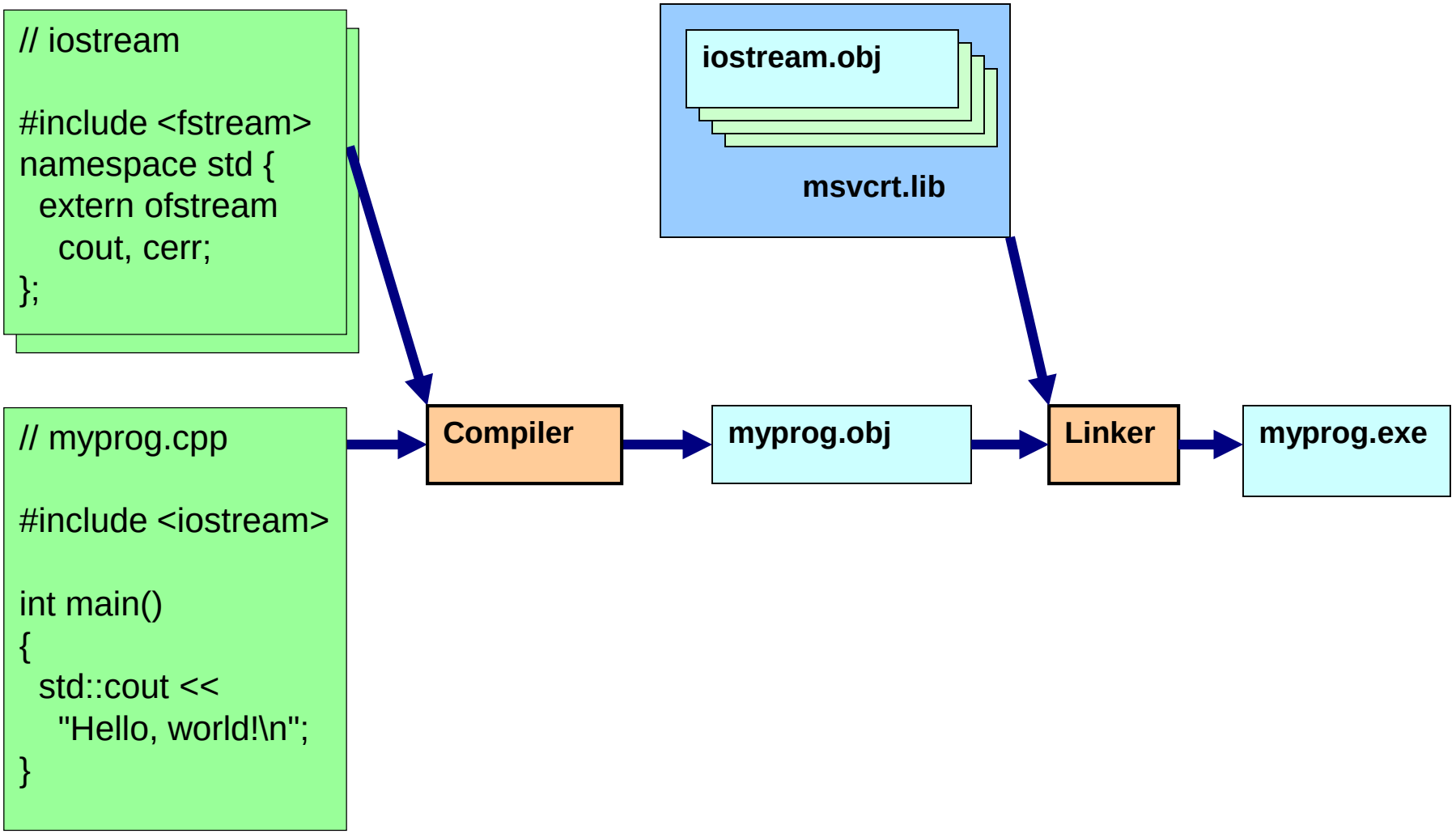
```cpp
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

using t_arg = std::vector< std::string>;
void world( const t_arg & arg);

#endif
```

```cpp
// main.cpp
#include "world.hpp"

int main( int argc, char * * argv)
{
  world( t_arg{ argv + 1, argv + argc});
  return 0;
}
```

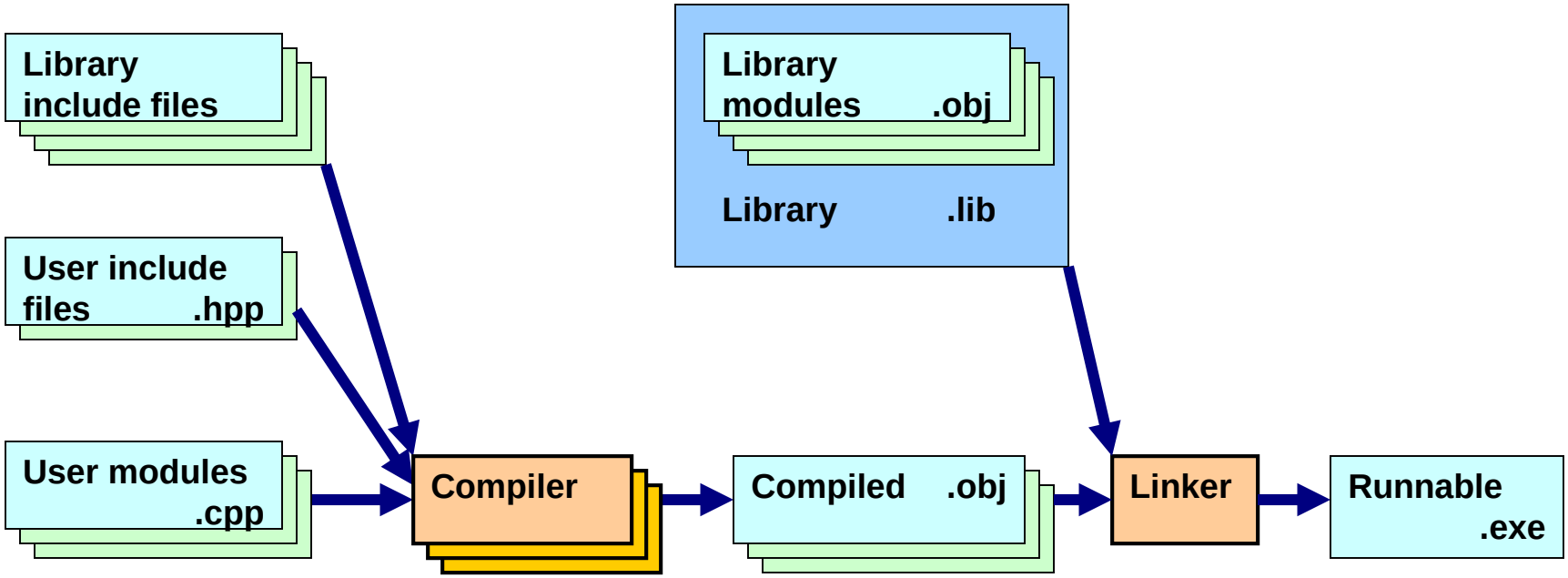
```cpp
// world.cpp
#include "world.hpp"
#include <iostream>

void world( const t_arg & arg)
{
  if ( arg.empty() )
  {
    std::cout << "Hello, world!"
      << std::endl;
  }
}
```

Compilation and linking

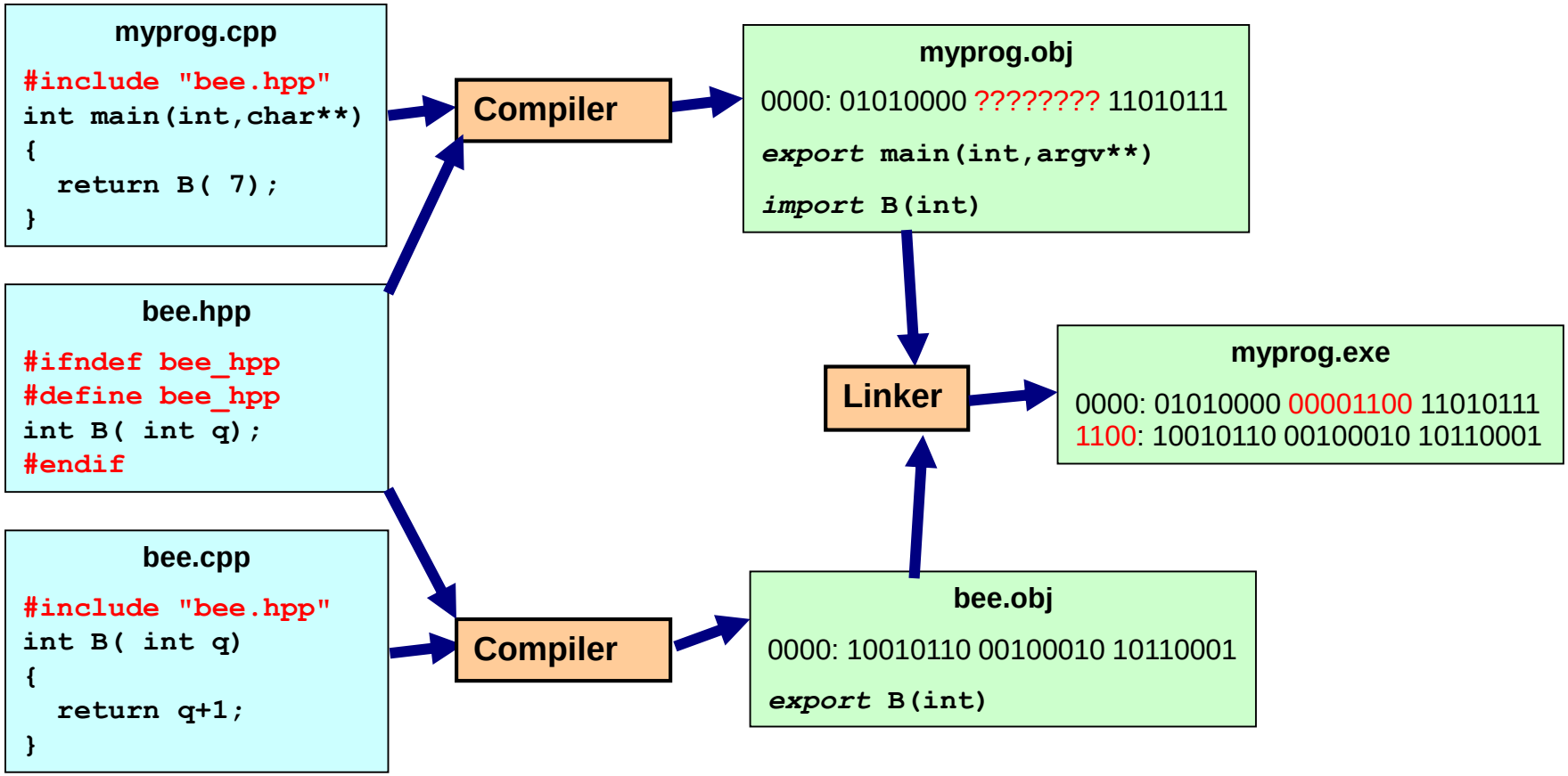
Single-module programs - static linking



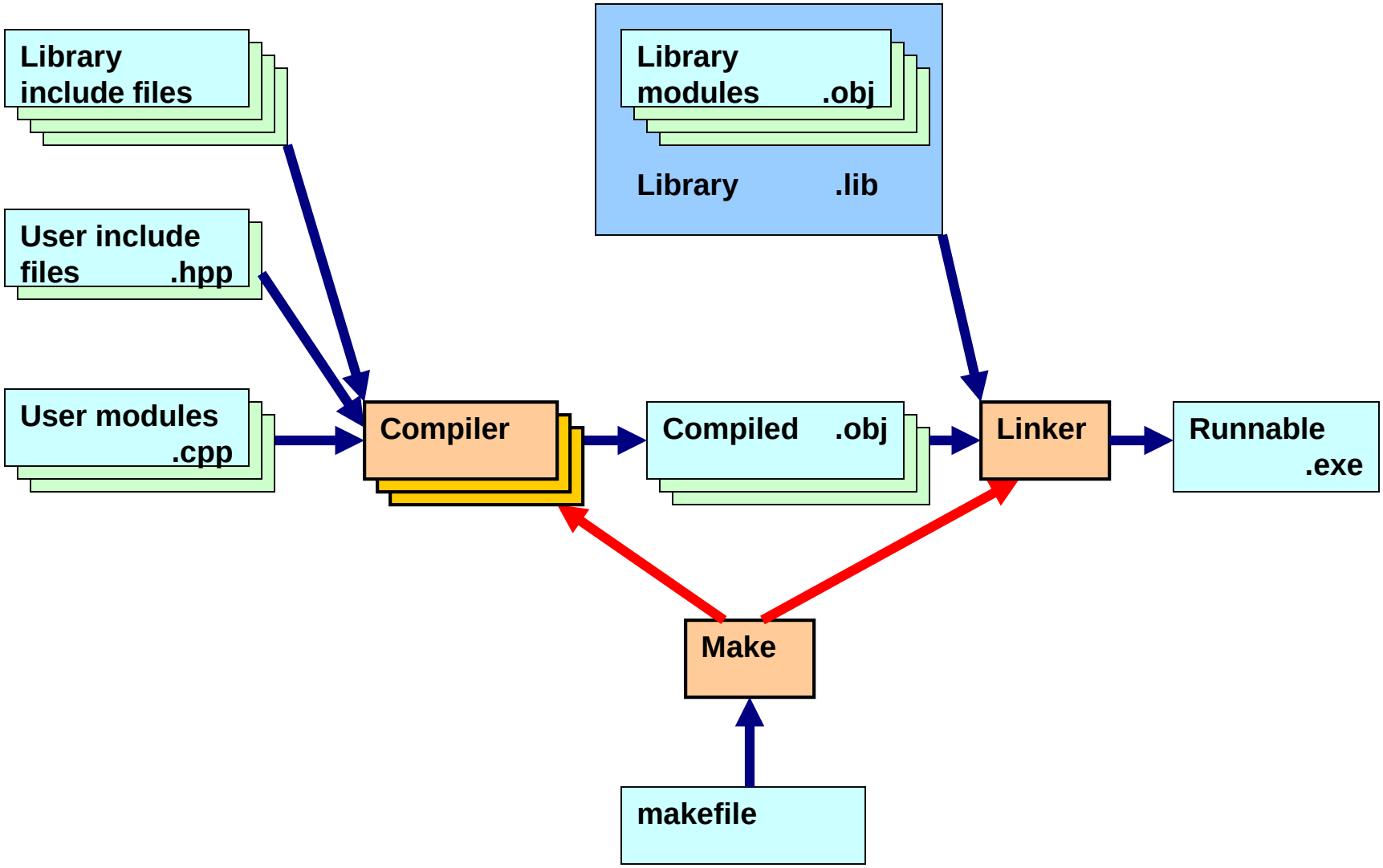
Multiple-module programs



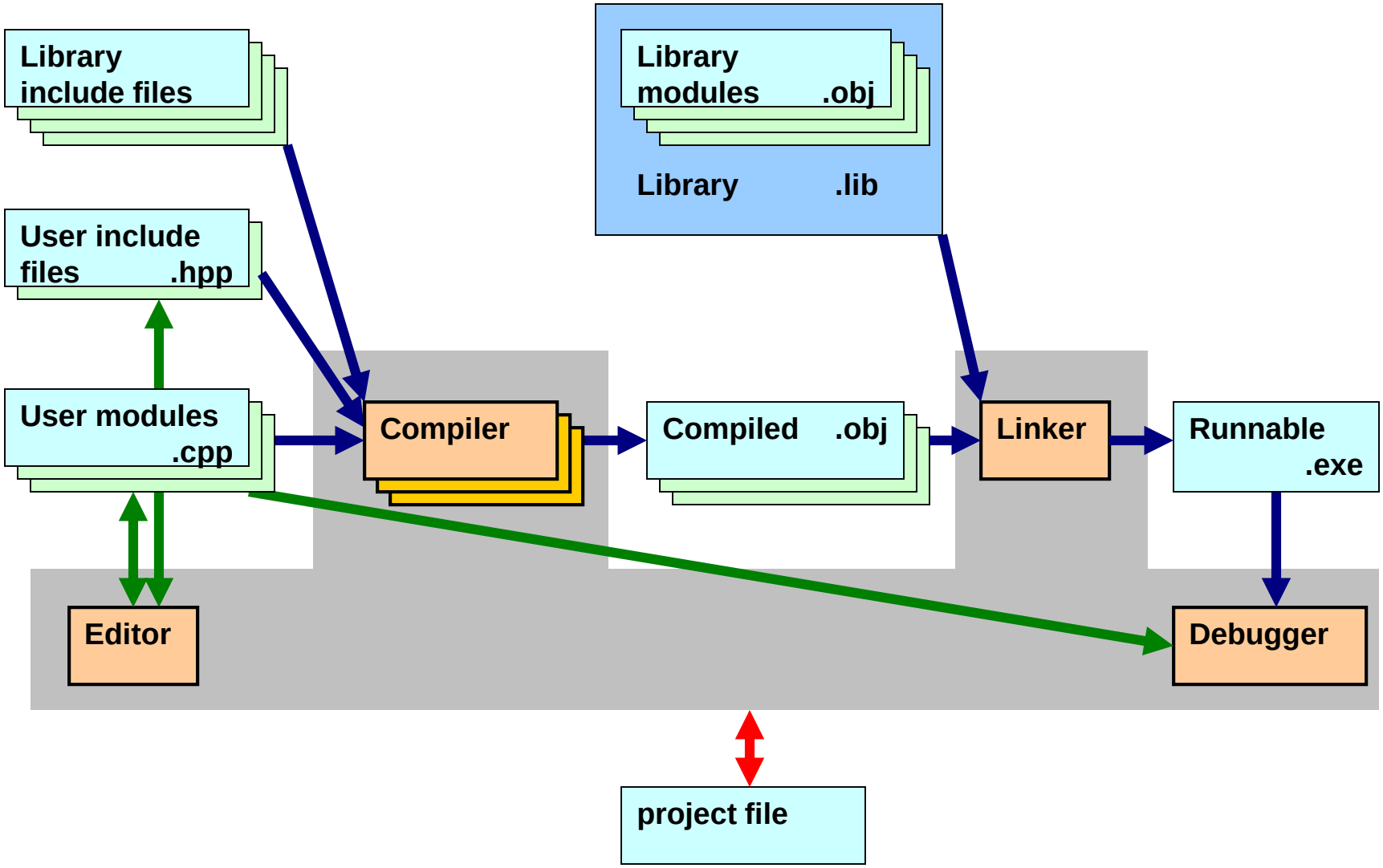
Module interfaces and linking



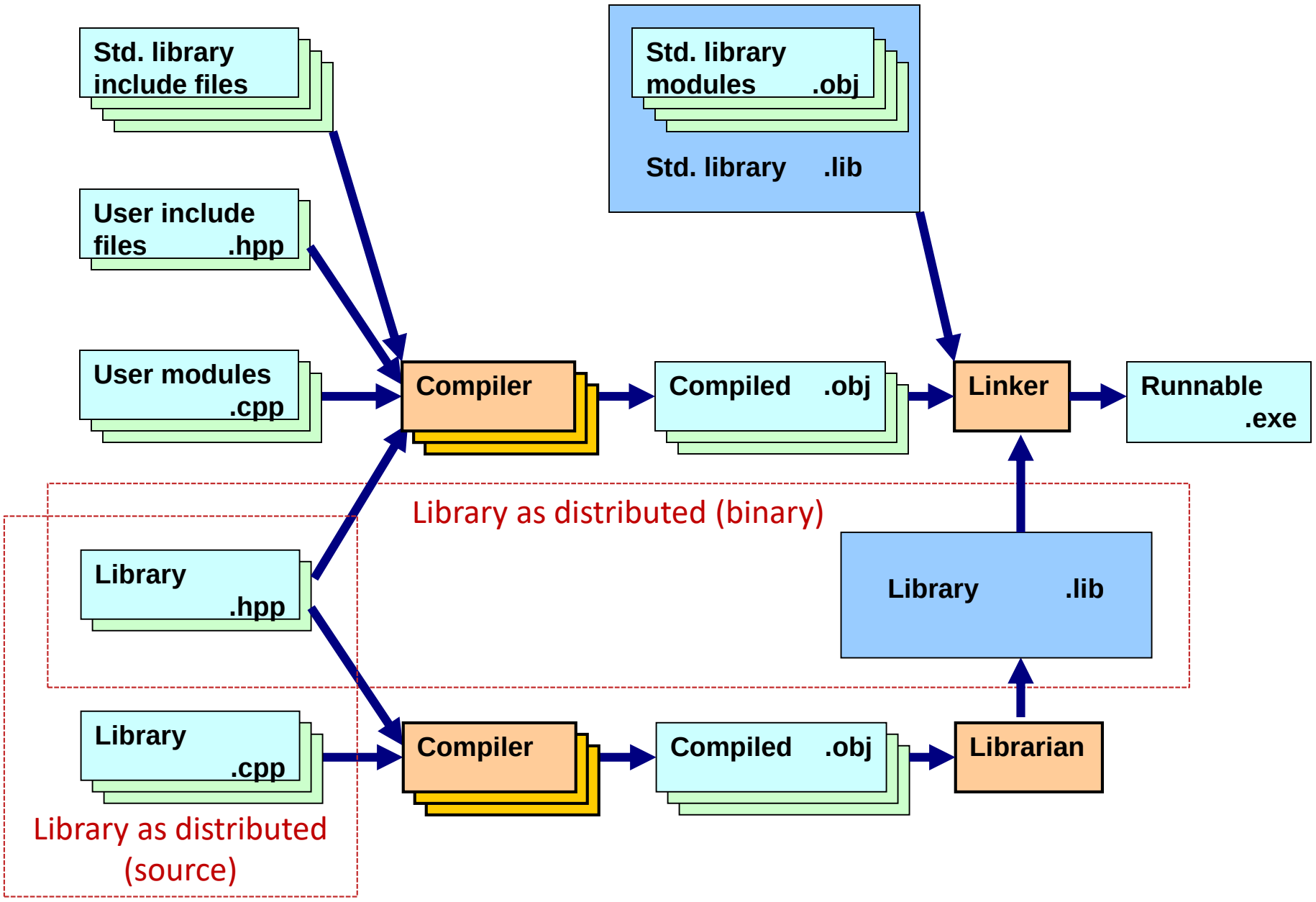
make


Library include files
User include files .hpp
User modules .cpp
Library modules .obj
Library .lib
Compiler
Compiled .obj
Linker
Runnable .exe
Make
makefile

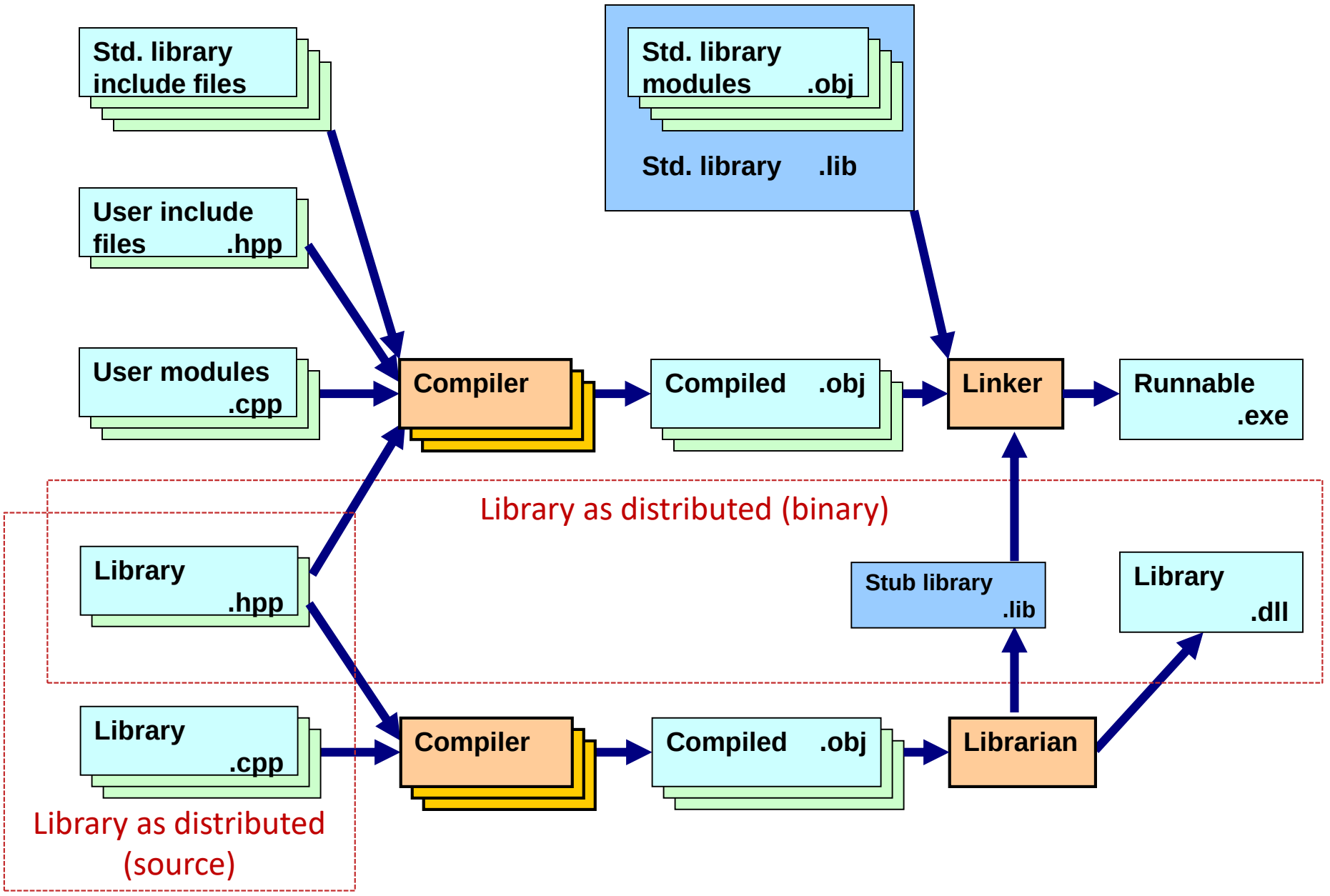
Integrated environment



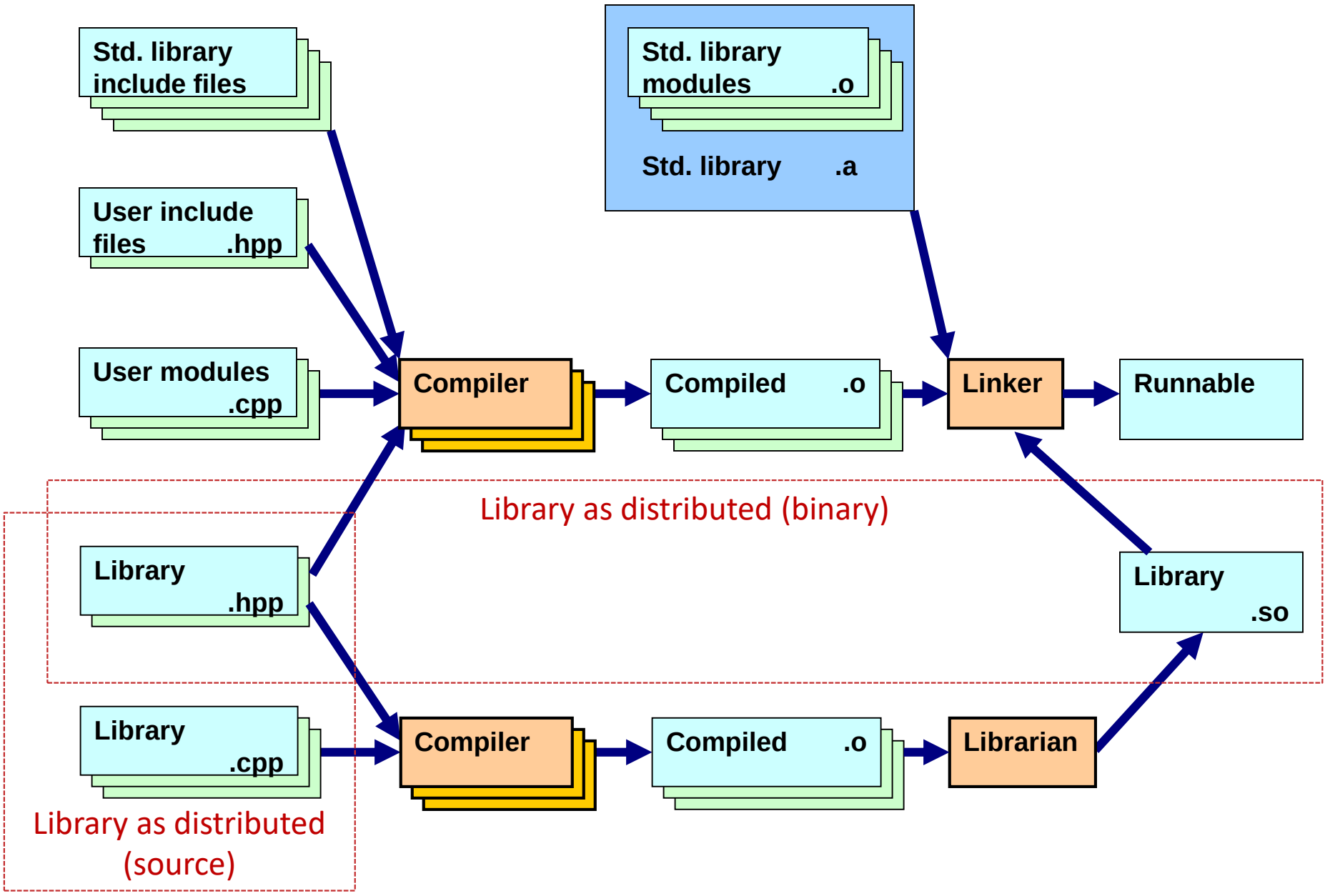
Static libraries



Dynamic libraries (Microsoft)



Dynamic libraries (Linux)



.cpp/.hpp - best practices

- .hpp – "header files"
  - Protect against repeated inclusion

```
#ifndef myfile_hpp_
#define myfile_hpp_
/* … */
#endif
```

  - Use include directive with double-quotes

```
#include "myfile.hpp"
```

    - Angle-bracket version is dedicated to standard libraries

```
#include <iostream>
```

  - Use #include only in the beginning of files (after ifndef+define)
  - Make header files independent: it must include everything what it needs

- .cpp - "modules"
  - Incorporated to the program using a project/makefile
    - Never include using #include

.cpp/.hpp - best practices

- .hpp – "header files"
  - Declaration/definitions of types and classes
  - Implementation of small functions
    - Outside classes, functions must be marked "inline"

```
inline int max( int a, int b) { return a > b ? a : b; }
```

  - Headers of large functions

```
int big_function( int a, int b);
```

  - Extern declarations of global variables

```
extern int x;
```

    - Consider using singletons instead of global variables
  - Any generic code (class/function templates)
    - The compiler cannot use the generic code when hidden in a .cpp

- .cpp - "modules"
  - Implementation of large functions
    - Including "main"
  - Definitions of global variables and static class data members
    - May contain initialization

```
int x = 729;
```

Dependences in code

- All identifiers must be declared prior to first use
  - Compilers read the code in one pass
  - Exception: Member-function bodies are analyzed at the end of the class
    - A member function body may use other members declared later
  - Generic code involves similar but more elaborate rules
- Cyclic dependences must be broken using declaration + definition

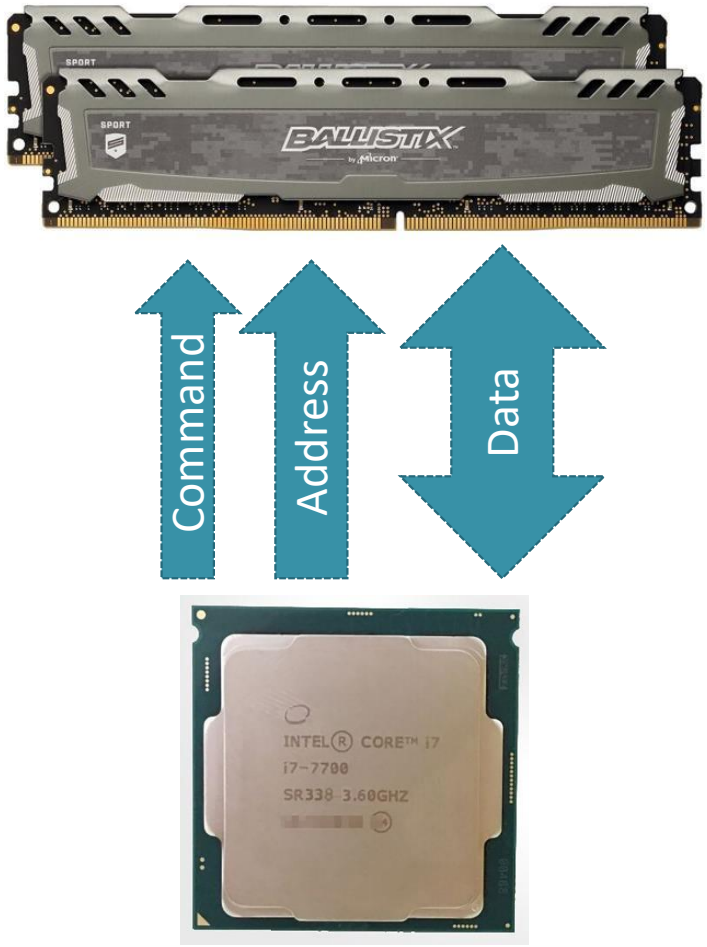
```
class one;                  // declaration
class two {
  std::shared_ptr< one> p_;
};
class one : public two      // definition
{};
```

  - Declared class is of limited use before definition
    - Cannot be used as base class, data-member type, in new, sizeof etc.

Všechno, co jste kdy chtěli vědět o Principech počítačů

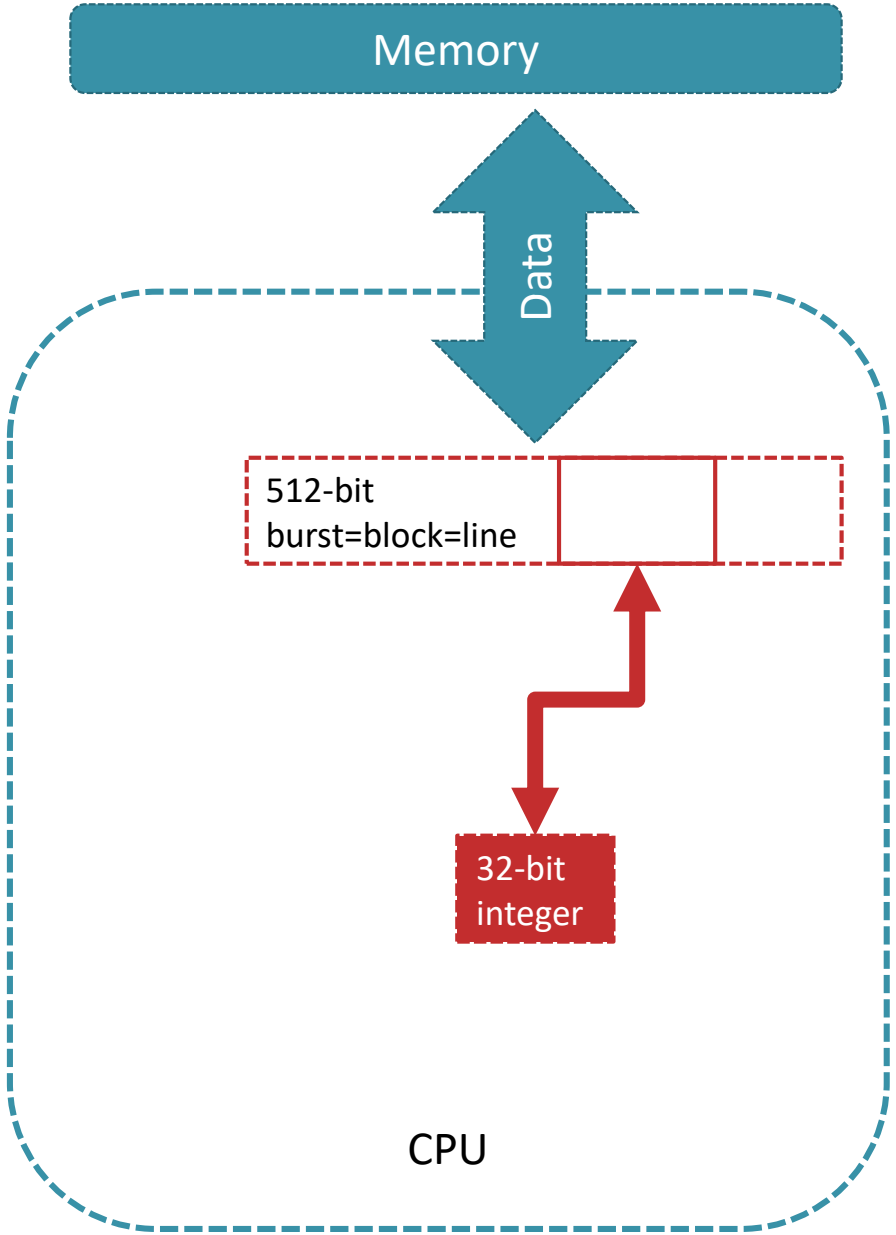
(ale báli jste se zeptat)

Minimal computer = CPU + RAM



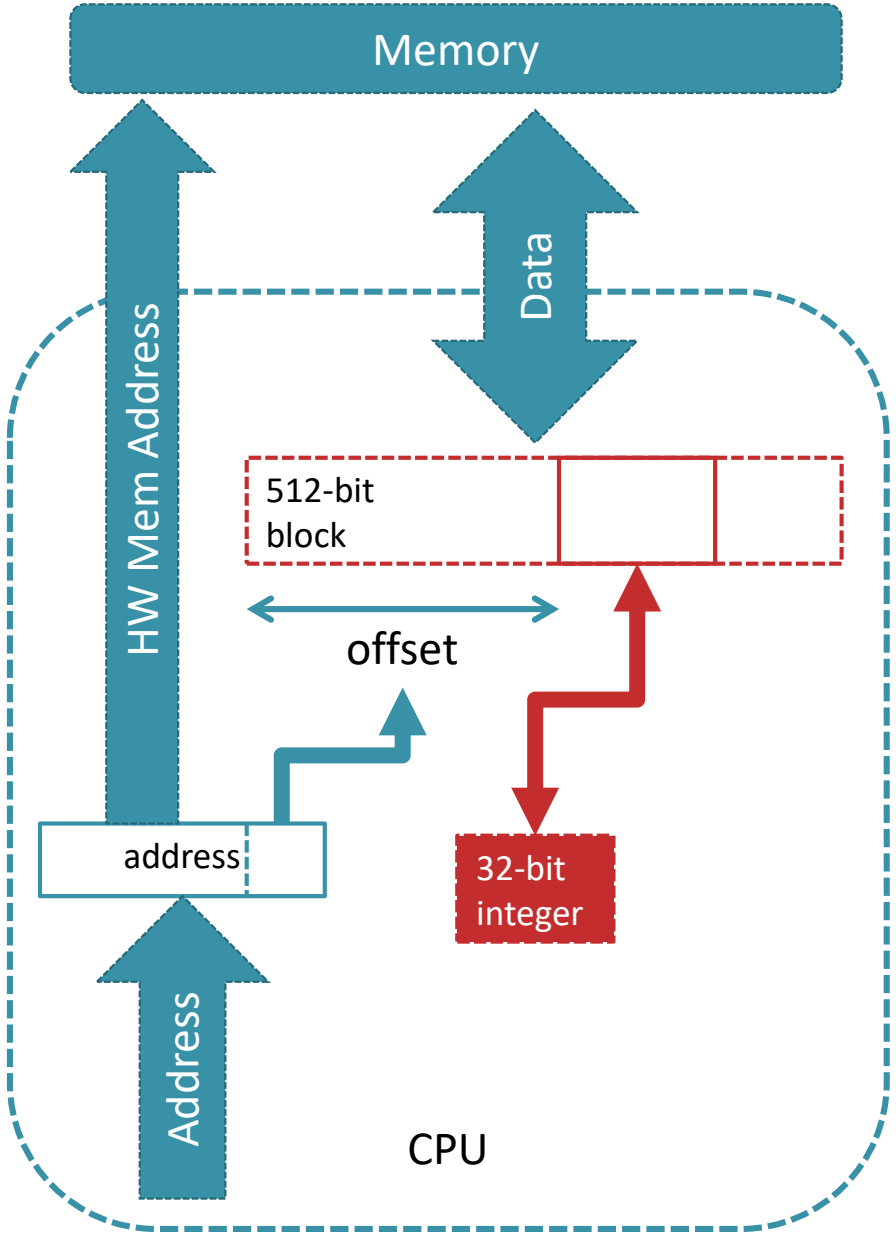
- CPU accesses memory using *commands*
  - Read/Write
  - Each command transfers a *burst* of data from/to memory
    - 512 bits in recent DDR4 chips
  - Address determines which 512-bit block is accessed
    - 16 GB = 256M * 512 bit
      - 28 address bits needed for 16 GB
    - Memory sizes are marketed in *bytes* although there are no byte-size elements in the hardware
  - The *address space* is not necessarily contiguous
    - Other types of memory (or I/O) may reside there
  - In reality, things are far more complicated

Inside CPU



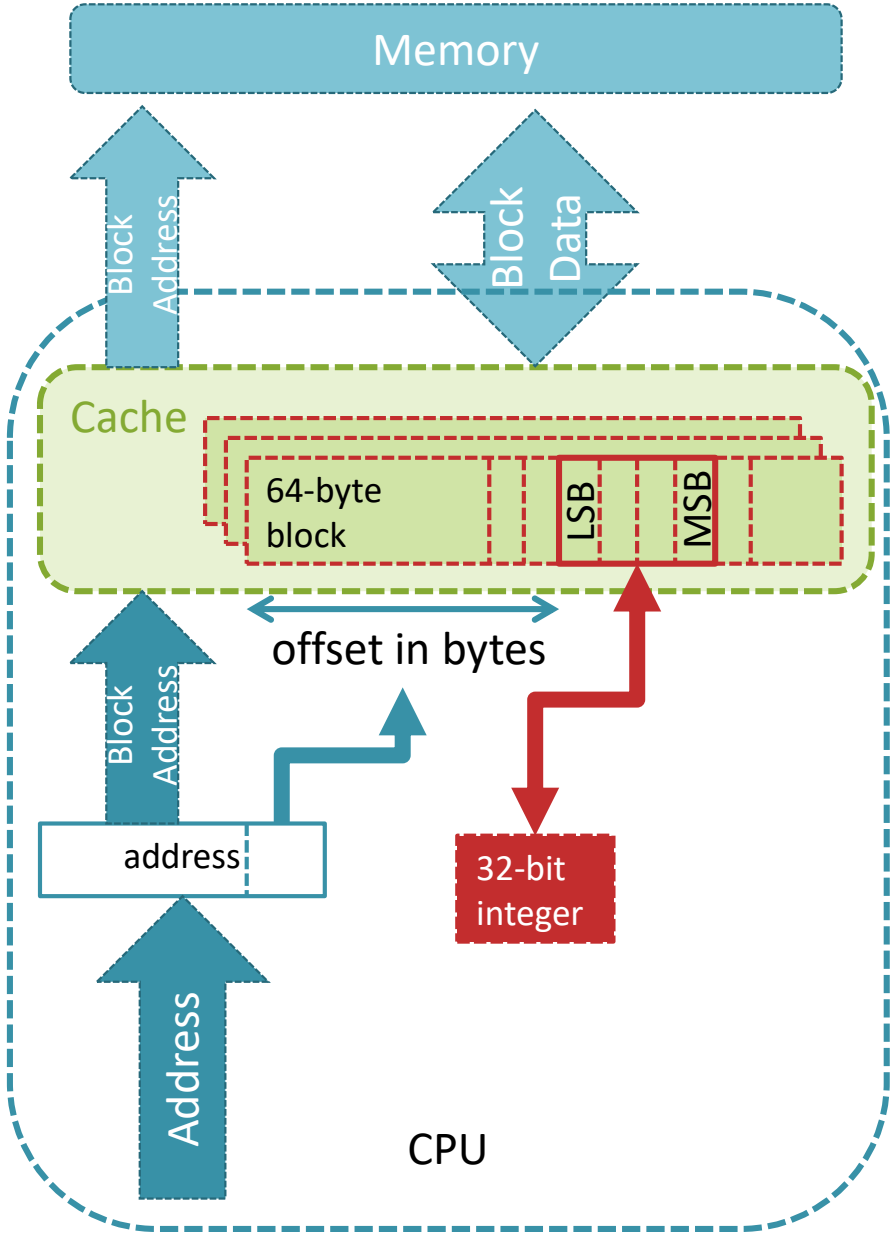
- CPU can work with *elementary data types*
  - Integers of various widths
    - today: 8, 16, 32, 64 bits
  - Floating-point in different formats
    - Intel/AMD: 32, 64, 80 bits
- CPU can read/write the elementary data types
  - only at some positions wrt. the 512-bit memory blocks
  - any elementary read/write requires reading the complete 512-bit block from the memory
    - or two blocks if across border
  - any elementary write results in writing the modified 512-bit block (or two) back to the memory
  - cache may reduce the number of block reads/writes if data in the same block are accessed
    - in cache, blocks are called *lines*

Inside CPU - addressing



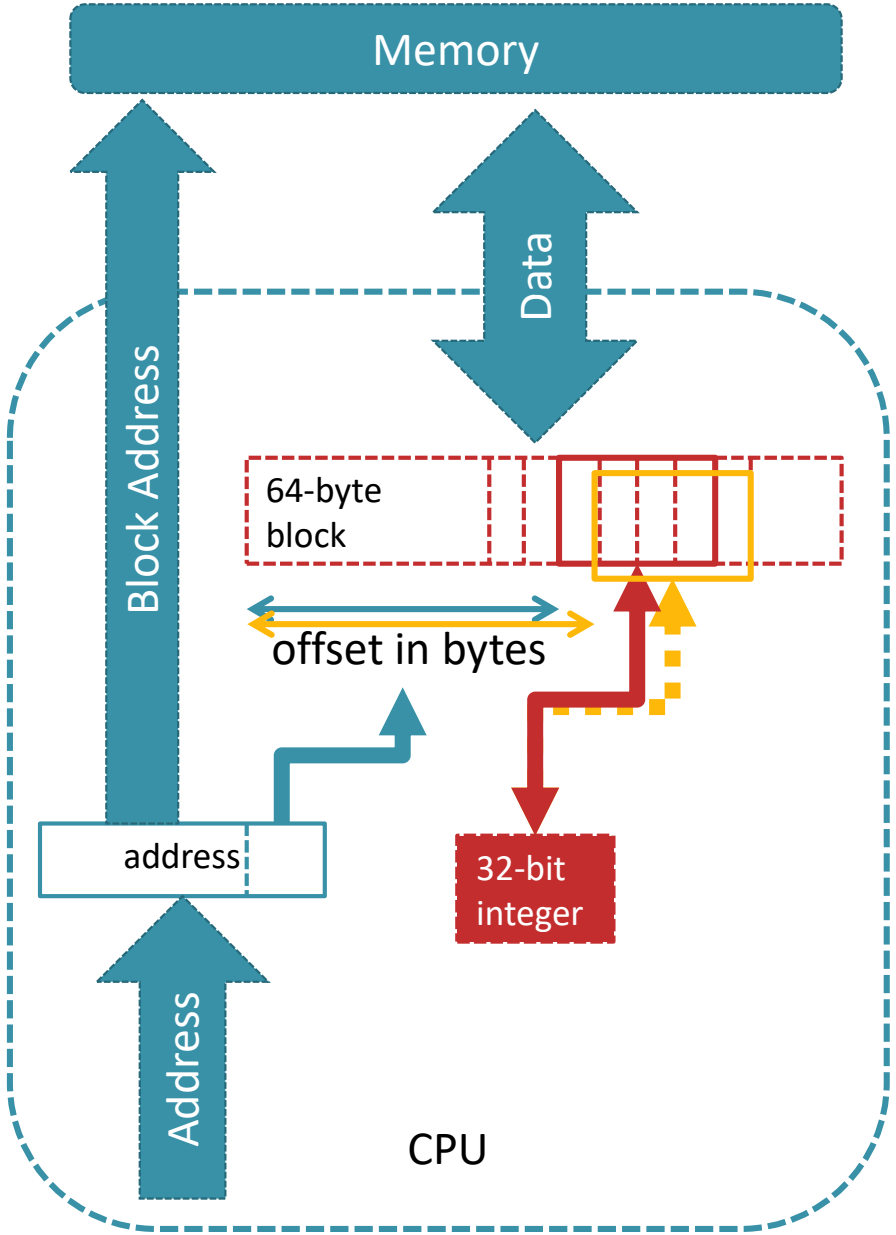
- CPU can internally read/write the elementary data types
  - the offset inside block is determined by some lower bits of the address
- Address granularity
  - how many bits are skipped when address is incremented by 1
  - 1 bit granularity is impractical
  - early computers used the size of their floating-point data type
    - often exotic values like 42 bits
  - the first C compiler targeted PDP-7
    - 18 bit address granularity
  - text processing requires addressing individual characters
    - 8 bit address granularity
    - first appeared in 1960's
    - other sizes died out in 1980's

Inside CPU – 8-bit address granularity



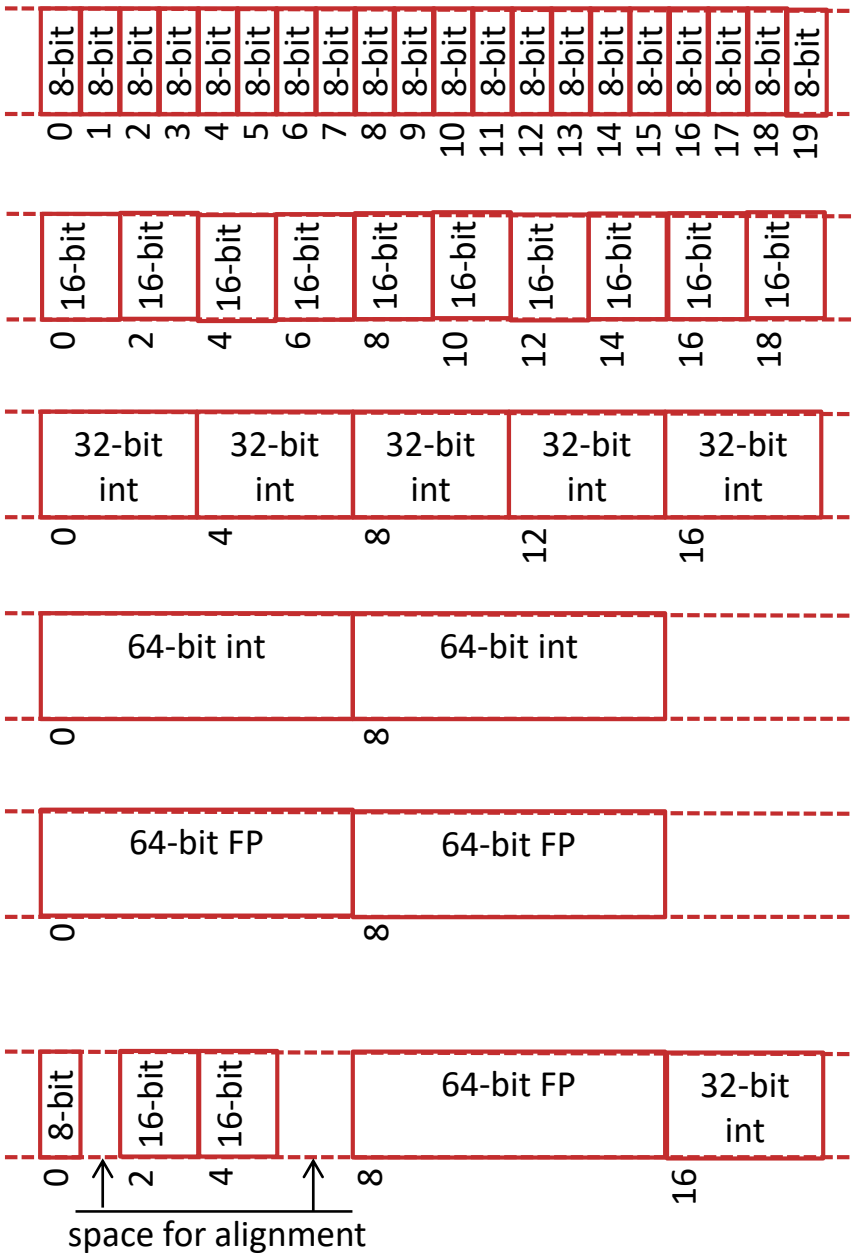
- 8 bit address granularity
  - no way to directly address bits
    - difficult implementation of bit arrays
    - no pointers/references to bits
  - most elementary data types span several bytes – order must be defined
    - Intel/AMD: *little-endian* – lower bits at lower addresses
    - software can see the order when accessing memory by bytes
  - order of bits inside a byte is irrelevant
    - software cannot see where bits are stored in the memory

Inside CPU – alignment



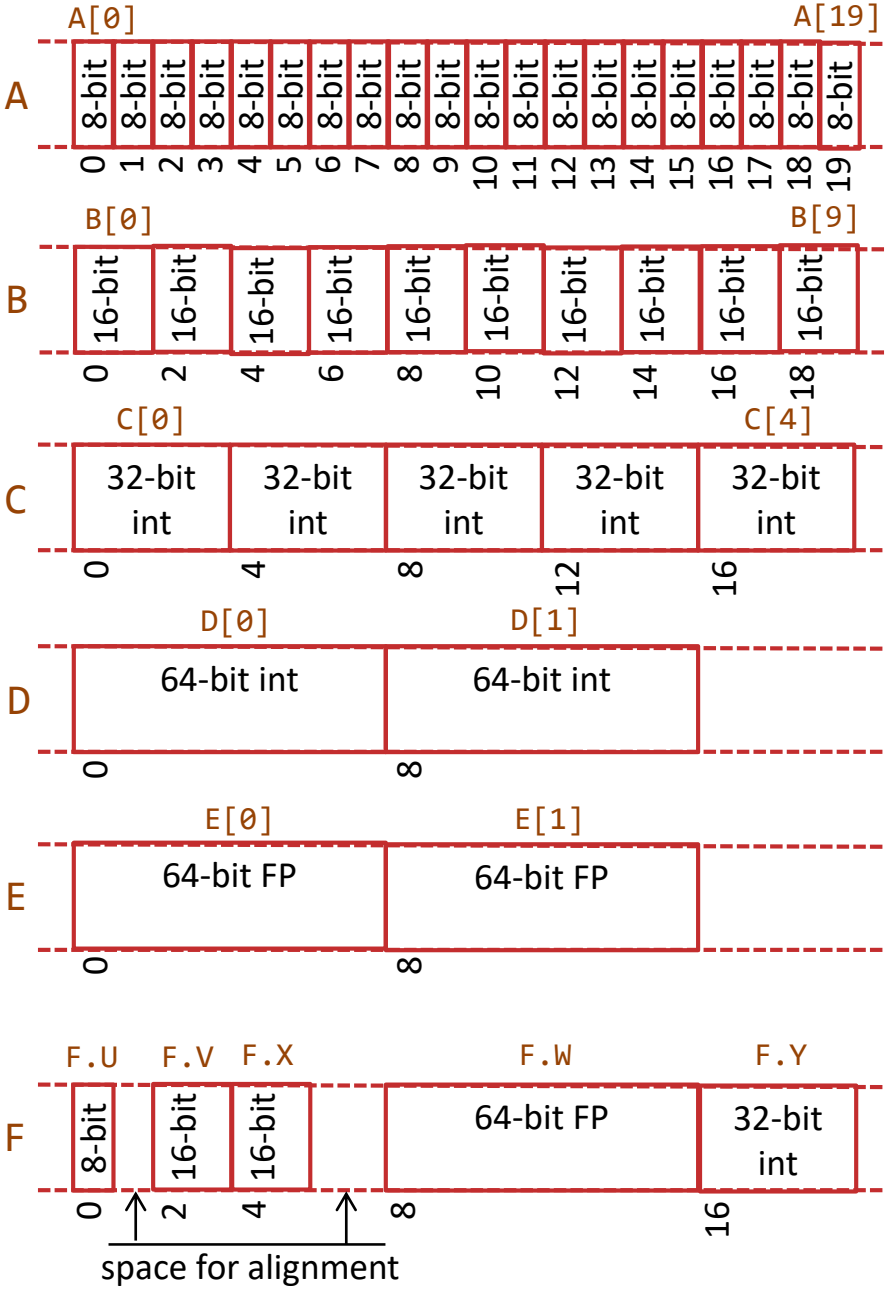
- CPU can read/write the elementary data types
  - only at some addresses wrt. the 64-byte memory blocks
  - *aligned* addresses: offset is a multiple of data type size
    - CPU contains hardware for quick access to these positions
  - *unaligned* addresses
    - Intel/AMD: access is possible at any byte-granular address but slower (emulated by hardware)
    - some platforms cannot access unaligned data at all (*fault*)

Data in memory – arrays and structures



- The memory does NOT store any type information
  - Memory is just an array of bits addressable at 8-bit boundaries
- The type is determined from the instruction which accesses the data
  - Only elementary types supported
- The compiler generates instructions with the required elementary type
  - Language rules try to maintain type safety
    - access type is derived from the type of the variable
    - data are read with the same elementary type as they were written
  - Low-level languages allow breaching of the type safety
    - when overridden using type cast
    - when something wrong happens
      - array overflows
      - access into deallocated data

Data in memory – arrays and structures in C++



- Arrays
  - N elements of the same type

```
std::int8_t A[20];
std::int16_t B[10];
std::int32_t C[5];
std::int64_t D[2];
double E[2];
```

- Structures
  - Several elements of different types
    - may contain gaps for alignment

```
struct S {
  std::int8_t U;
  std::int16_t V;
  double W;
  std::int16_t X;
  std::int32_t Y;
};
S F;
```

Signed integer types – names in C/C++

Signed integer types Number of bits	Not guaranteed to exist (on exotic architectures)	Not guaranteed to exactly match (on exotic architectures)	x86 (32-bit)	x64 (64-bit)	
			MS Visual C++		GNU C++
8	`std::int8_t`	`std::int_least8_t`	`signed char` `std::int_fast8_t`		
16	`std::int16_t`	`std::int_least16_t`	`[signed] short [int]`		
32	`std::int32_t`	`std::int_least32_t`	`[signed] int`		

			`[signed] long [int]`
			`std::int_fast16_t`
			`std::int_fast32_t`		

			`std::ptrdiff_t`		
64	`std::int64_t`	`std::int_least64_t`	`[signed] long long [int]` `std::int_fast64_t`		

					`[signed] long [int]`
					`std::int_fast16_t`
					`std::int_fast32_t`

				`std::ptrdiff_t`	

- Built-in types are denoted using a sequence of keywords
  - [some keywords are optional]
- Library types are denoted using an identifier
  - C: #include <stdint.h>
  - C++: #include <cstdint> and use namespace prefix std::

Signed integer types – names in C/C++

Signed integer types Number of bits	Not guaranteed to exist (on exotic architectures)	Not guaranteed to exactly match (on exotic architectures)	x86 (32-bit) / MS Visual C++	x64 (64-bit) / GNU C++
8	`std::int8_t`	`std::int_least8_t`	`signed char` `std::int_fast8_t`	
16	`std::int16_t`	`std::int_least16_t`	`[signed] short [int]`	
32	`std::int32_t`	`std::int_least32_t`	`[signed] int`	

			`[signed] long [int]`
			`std::int_fast16_t`
			`std::int_fast32_t`	

			`std::ptrdiff_t`	
64	`std::int64_t`	`std::int_least64_t`	`[signed] long long [int]` `std::int_fast64_t`	

				`[signed] long [int]`
				`std::int_fast16_t`
				`std::int_fast32_t`

				`std::ptrdiff_t`

- char/short/int/long/long long – size depends on architecture and compiler
- std::int{N}_t has exactly N bits (not guaranteed to exist)
- std::int_least{N}_t is the smallest type that has at least N bits
- std::int_fast{N}_t is the fastest type that has at least N bits
- std::ptrdiff_t has enough bits to store indexes to any array that fits in memory

Unsigned integer types – names in C/C++

<table>
<tr><td>Unsigned integer types
Number of bits</td><td>Not guaranteed to exist (on exotic architectures)</td><td>Not guaranteed to exactly match (on exotic architectures)</td><td colspan="2">x86 (32-bit)
MS Visual C++</td><td colspan="2">x64 (64-bit)
GNU C++</td></tr>
<tr><td>8</td><td>std::uint8_t</td><td>std::uint_least8_t</td><td colspan="4">unsigned char
std::uint_fast8_t</td></tr>
<tr><td>16</td><td>std::uint16_t</td><td>std::uint_least16_t</td><td colspan="4">unsigned short [int]</td></tr>
<tr><td rowspan="2">32</td><td rowspan="2">std::uint32_t</td><td rowspan="2">std::uint_least32_t</td><td colspan="4">unsigned [int]</td></tr>
<tr><td colspan="2">unsigned long [int]
std::uint_fast16_t
std::uint_fast32_t</td><td colspan="2"></td></tr>
<tr><td rowspan="2">64</td><td rowspan="2">std::uint64_t</td><td rowspan="2">std::uint_least64_t</td><td colspan="4">unsigned long long [int]
std::uint_fast64_t</td></tr>
<tr><td colspan="2"></td><td colspan="2">unsigned long [int]
std::uint_fast16_t
std::uint_fast32_t</td></tr>
<tr><td colspan="3"></td><td colspan="4">std::size_t</td></tr>
</table>

- All integer types have unsigned versions
  - Use the unsigned keyword for built-in types
  - Use uint instead of int in library names
  - Use size_t instead of ptrdiff_t

Signed vs. unsigned

- Range of n-bit integer types
  - signed: $-2^{(n-1)} .. 2^{(n-1)}-1$
  - unsigned: $0 .. 2^n-1$
- Arrays are always indexed as 0 .. S-1
  - The unsigned type std::size_t is large enough for all in-memory arrays and containers
- Do we really need signed integer types?
  - Is there a real-world situation where numbers are negative but not fractional?
    - Floor numbers? But which one is "1"?
  - There are few in engineering: Number of steps to rotate a servomotor...
  - There is one important case in programming: Difference of two indexes

```
delta = index1 - index2;
```

    - Although indexes are usually unsigned, the difference may be negative
    - In C/C++, we may also compute difference of two pointers (or iterators)

```
char * ptr1 = /*...*/; char * ptr2 = /*...*/; delta = ptr1 - ptr2;
```

    - Declare the variable delta as std::ptrdiff_t in both cases

Which integer type?

- If the values are often passed to or returned from a library
  - use the same type as the library uses
    - Example: File sizes may not fit in std::size_t in a 32-bit program. Use the type returned from the function you use to measure the file size.
    - Example: Intel MKL (Math Kernel Library) can do matrix operations. The sizes of the matrices are passed as type MKL_INT. If you frequently call MKL functions, use MKL_INT for variables holding matrix sizes. If you use std::size_t (as recommended in general), compilers may issue warnings on every MKL call.
- if the data have to match a predefined binary format
  - e.g. in a binary file or a network packet
    - Note: you will likely need a type cast (reinterpret_cast) when reading/writing/sending/receiving the binary data
  - use std::[u]int{N}_t
    - these types do not exist on exotic platforms – but the required file/network library will likely be missing too
- if you need to save space
  - and you are sure about the range of data for any foreseeable future
    - "640KB ought to be enough for anybody"
  - use std::[u]int_least{N}_t
    - use the corresponding std::[u]int_fast{N}_t for local variables if you perform non-trivial math on them
- if differences are stored in the variable
  - use std::ptrdiff_t
- otherwise
  - your variables will likely serve as indexes or sizes of arrays/containers
  - use std::size_t
    - it will scale down if you compile for a 32-bit platform
    - it will automatically scale up if you ever have more than 16 exabytes of memory
- Note: Use of built-in types (int) is recommended only if forced by someone else's mistake.

Floating point types

Number of bits (Intel/AMD)	MS Visual C++	GNU C++
float	32	32
double	64	64
long double	80	128 (due to alignment)

- Floating-point type names have only keyword forms
- Format of floating-point types is not defined by the C++ standard
  - Most implementations use IEEE-754 for float and double
  - May support special values
    - infinity
    - NaN (not-a-number)
- Standard library template std::numeric_limits reports the properties of integer and floating point types
  - Compile-time system (a.k.a. *traits*) – may return compile-time constants

```
#include <limits>

static constexpr bool INF = std::numeric_limits<float>::has_infinity;

static constexpr int M = std::numeric_limits<float>::max_exponent10;

static_assert( INF && M >= 38, "At least IEEE-754 required");
```

    - static_assert triggers compile-time error if the condition is false
      - the condition must be compile-time constant (constexpr)

```
static constexpr float INFTY = std::numeric_limits<float>::infinity();
```

Enumeration types

- Enumeration declaration declares
  - a new type (optional)
  - a set of constants (optional)
    - by default, value is previous_constant+1 (0 if no previous)
- Unscoped enumeration [enum]
  - constants are directly visible
  - implicitly convertible to integral types
- Scoped enumeration [enum class]
  - constants must be accessed by qualified name enum_type::enum_constant
  - no implicit conversions (but may be converted using a type cast)
- Underlying type
    - implementation-defined size (large enough to fit all named constants)

```
enum Foo { a, b, c = 10, d, e = 1, f, g = f + c };          // unscoped, a = 0, b = 1, d = 11, f = 2, g = 12
```

    - explicitly defined underlying type (enforces unsignedness and small size)

```
enum class Bar : std::uint_least8_t { max = 255, min = 0 }; // scoped, Bar::max, Bar::min
```

- Extreme cases
    - declaring constants

```
enum { N = 100 };
```

        - no-longer in use – constexpr is preferred (allows specifying type)

```
static constexpr std::size_t N = 100;
```

    - a new elementary type, formally different from the original

```
enum class another_int : int;
```

Boolean

- bool
  - false, true
    - implicit conversion to integer types produces 0, 1
  - implicit conversion from numeric, unscoped enumeration, and pointer types
    - produces true if non-zero (non-null)
    - this conversion may be enforced using double negation (!! e)
    - many library-defined types allow such conversion too

```
std::ifstream F( "file.txt"); bool success = F;
```

  - produced by relational operators ==, !=, <, <=, >, >=
  - consumed by conditional expression (?:), Boolean AND (&&), Boolean OR (||)
    - short-circuit evaluation

```
while ( p && p->v != x ) p = p->next;
```

      - p->v is not evaluated if p is nullptr
    - never use bitwise AND (&), bitwise OR (|) on bool
  - consumed by if, while, for
  - occupies a byte = 8 bits (except exotic hardware)
    - vector<bool> uses 1 bit per element

Character types – names in C/C++

Encoding supported		MS Visual C++	GNU C++
7-bit ASCII 8-bit code page (UTF-8)	`char`		
UCS-2 (UTF-16)	`char16_t`	`wchar_t`	
UCS-4 = UTF-32	`char32_t`		`wchar_t`

- All character type names are keywords
- The type does not imply the character encoding used
- Character types support integer arithmetic operations directly

```
char ch = /* ... */; if ( ch >= '0' && ch <= '9' ) { std::int_least8_t value = ch - '0'; /* ... */ }
```

  - Formally distinct but compatible with integer types
  - Beware: Implementation-defined signedness
  - Beware: Comparison is binary, not alphabetic
- Fixed-length encodings
  - one character encoded by one element of the corresponding datatype
    - char: 7/8-bit encoding; code page is implementation-defined
    - char16_t: UCS-2 encoding of a subset of Unicode
    - char32_t: UTF-32 (equivalent to UCS-4) encoding of Unicode
- Variable-length encodings
  - one character encoded by one or more elements of the corresponding datatype
    - char: UTF-8 encoding of Unicode
    - char16_t: UTF-16 encoding of Unicode
  - the only operations supported by C++ standard are conversions to/from fixed-length encodings

String types in C/C++

Encoding supported	C/C++	C++

7-bit ASCII
8-bit code page
(UTF-8)
	`char[N]`
	`const char*`
	`char*` 
		`std::string`

UCS-2
(UTF-16)
	`char16_t[N]`
	`const char16_t*`
	`char16_t*` 
		`std::u16string`

UCS-4 = UTF-32	`char32_t[N]`
	`const char32_t*`
	`char32_t*` 
		`std::u32string`

implementation defined
(16/32 bit)
	`wchar_t[N]`
	`const wchar_t*`
	`wchar_t*` 
		`std::wstring`

- String types are not elementary types
  - An illusion created by conventions, the compiler and/or the standard library
- C representation: string is an array of characters
  - String length indicated by terminating zero
    - char[10] supports up to 9 characters only
  - Passed to functions as a pointer to the first element (C/C++ rule for all naked arrays)
    - the pointer does not indicate the size of the underlying memory buffer
  - ***NEVER* TRY TO ASSIGN/EXTEND/CONCATENATE STRINGS IN ANY OF THEIR C-REPRESENTATIONS, REALLY *NEVER***
    - **Those who tried are responsible for a majority of bugs, crashes, exploits, million dollar losses, ...**
    - **Consequence: Never use pure C for anything that works with strings**
  - Reading C-style strings is safe

String types in C/C++

Encoding supported	C/C++	C++	literals

7-bit ASCII
8-bit code page
(UTF-8)
	`char[N]`
	`const char*`
	`char*` 
		`std::string`	`"Hello, ASCII!"`
			`"Čau, cp-1250?"`
			`u8"Tschüß, UTF-8!"`

UCS-2
(UTF-16)
	`char16_t[N]`
	`const char16_t*`
	`char16_t*` 
		`std::u16string`	`u"Tschüß, UTF-16!"`

UCS-4 = UTF-32	`char32_t[N]`
	`const char32_t*`
	`char32_t*` 
		`std::u32string`	`U"Tschüß, UTF-32!"`

implementation defined
(16/32 bit)
	`wchar_t[N]`
	`const wchar_t*`
	`wchar_t*` 
		`std::wstring`	`L"Tschüß, was?"`

- C++ representation: string is a standard library container
  - Similar to std::vector
  - Works as value type – deep copying
  - Dynamically allocated - supports assignment/extension/concatenation safely
  - Terminating zero is not visible, not included in size()
  - Implicit conversion from C-style strings
  - c_str(): Explicit conversion to read-only C-style string
  - Rather unusable for variable-length encodings
- String constants are represented by read-only character arrays (as in C)
  - When passed further as C++ string, dynamic allocation and copying occurs

String types in C/C++

Encoding supported	C/C++	C++	C++17 „readonly string“

7-bit ASCII
8-bit code page
(UTF-8)
	`char[N]`
	`const char*`
	`char*` 
		`std::string`	`std::string_view`

UCS-2
(UTF-16)
	`char16_t[N]`
	`const char16_t*`
	`char16_t*` 
		`std::u16string`	`std::u16string_view`

UCS-4 = UTF-32	`char32_t[N]`
	`const char32_t*`
	`char32_t*` 
		`std::u32string`	`std::u32string_view`

implementation defined
(16/32 bit)
	`wchar_t[N]`
	`const wchar_t*`
	`wchar_t*` 
		`std::wstring`	`std::wstring_view`

- string_view is a reference to the contents of a std::string or a C-string
  - Does not participate in ownership
  - Does not prevent destruction/modification of the string referred to
- Passing strings into functions
  - Before C++17 – pass std::string by reference

```
void f( const std::string & s);
```

  - After C++17 – pass std::string_view by value

```
void f( std::string_view s);
```

    - Advantage: Does not involve copying if the actual argument is a C-string (e.g. a literal)

String types in C/C++

Encoding supported	C/C++	C++	C++17 „readonly string“

7-bit ASCII
8-bit code page
(UTF-8)
	`char[N]`
	`const char*`
	`char*` 
		`std::string`	`std::string_view`

UCS-2
(UTF-16)
	`char16_t[N]`
	`const char16_t*`
	`char16_t*` 
		`std::u16string`	`std::u16string_view`

UCS-4 = UTF-32	`char32_t[N]`
	`const char32_t*`
	`char32_t*` 
		`std::u32string`	`std::u32string_view`

implementation defined
(16/32 bit)
	`wchar_t[N]`
	`const wchar_t*`
	`wchar_t*` 
		`std::wstring`	`std::wstring_view`

- string_view is a reference to the contents of a std::string or a C-string
  - Does not participate in ownership – does not prevent destruction/modification of the string referred to
- Returning strings from functions
  - If the function computes the return value – ALWAYS return std::string BY VALUE

```
std::string f() { return “Hello ” + name; }
```

  - If the function returns a reference to a string stored elsewhere
    - to allow modification - return std::string by reference

```
std::string & f() { return object->name_; }
```

    - for read-only access - return std::string by const reference

```
const std::string & f() { return object->name_; }
```

    - C++17 - for TEMPORARY read-only access – return std::string_view by value

```
std::string_view f() { return object->name_; }
```

Returning references – FATAL MISTAKES

Reference to string

```
std::string & g()
{
  std::string local = “Hello”;
  return local;
}
```

```
const std::string & g()
{
  return “Hello”;
}
```

```
std::string global = “bflm”;
const std::string & g()
{
  return global + “.txt”;
}
```

Reference to the contents

```
std::string_view g()
{
  std::string local = “Hello”;
  return local;
}
```

- This is correct:

```
std::string_view g()
{
  std::string_view local = “Hello”;
  return local;
}
```

C-style

```
const char * g()
{
  std::string local = “Hello”;
  return local.c_str();
}
```

- This is correct:

```
const char * g()
{
  const char * local = “Hello”;
  return local;
}
```

Passing references as arguments – the broken cases

Reference to string

```
std::vector< std::string>
  global_v = { “Hello” };


void g()
{
  f(global_v[0]);
}


void f( const std::string & s)
{
  std::cout << s;          // OK

  global_v.clear();

  std::cout << s;          // CRASH
}
```

Reference to the contents

```
std::string
  global_s = “Hello”;


void g()
{
  f(global_s);
}


void f( std::string_view s)
{
  std::cout << s;          // OK

  global_s = “Welcome”;

  std::cout << s;          // CRASH
}
```

C-style

```
std::string
  global_s = “Hello”;


void g()
{
  f(global_s.c_str());
}


void f( const char * s)
{
  std::cout << s;          // OK

  global_s = “Welcome”;

  std::cout << s;          // CRASH
}
```

Returning references – the broken cases

Reference to string

```
std::vector< std::string>
  global_v = { “Hello” };


std::string & g()
{
  return global_v[0];
}


void f()
{
  std::string & s = g();
  std::cout << s;          // OK

  global_v.clear();        

  std::cout << s;          // CRASH
}
```

Reference to the contents

```
std::string
  global_s = “Hello”;


std::string_view g()
{
  return global_s;
}


void f()
{
  std::string_view s = g();
  std::cout << s;          // OK

  global_s = “Welcome”;    

  std::cout << s;          // CRASH
}
```

C-style

```
std::string
  global_s = “Hello”;


const char * g()
{
  return global_s.c_str();
}


void f()
{
  const char * s = g();
  std::cout << s;          // OK

  global_s = “Welcome”;    

  std::cout << s;          // CRASH
}
```

Returning references – the broken cases without global variables

Reference to string

```
class C { public:
  std::string & get_s() {
    return m_v[0];
  }
  void clear() {
    m_v.clear();
  }
private:
  std::vector< std::string>
    m_v = { “Hello” };
};
void f() {
  C obj;
  std::string & s = obj.get_s();
  std::cout << s;          // OK
  obj.clear();
  std::cout << s;          // CRASH
}
```

Reference to the contents

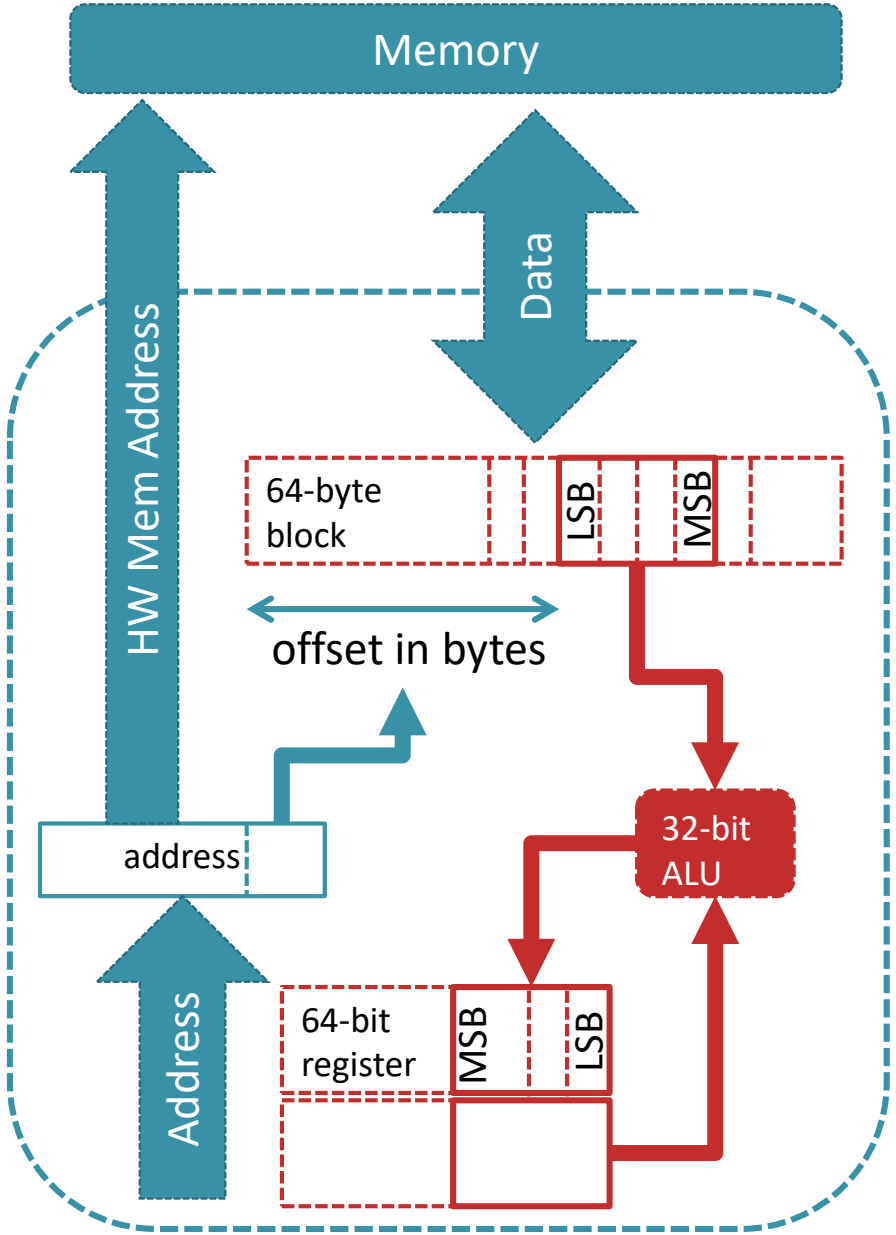
```
class C { public:
  std::string_view get_s() {
    return m_s;
  }
  void set_s( std::string_view p) {
    m_s = p;
  }
private:
  std::string m_s = “Hello”;
};
void f() {
  C obj;
  std::string_view s = obj.get_s();
  std::cout << s;          // OK
  obj.set_s( “Welcome”);
  std::cout << s;          // CRASH
}
```

- There is no bad code in the class C
  - Returning references is dangerous...
  - ... but speed matters
- The bad code is in f()
  - Storing a reference for some (long) time...
    - in the variable s
  - ... may be acceptable...
    - speed matters
  - ... but must be verified
    - avoid any changes in the same object
    - you never know what the methods really do

Všechno, co jste kdy chtěli vědět o Principech počítačů

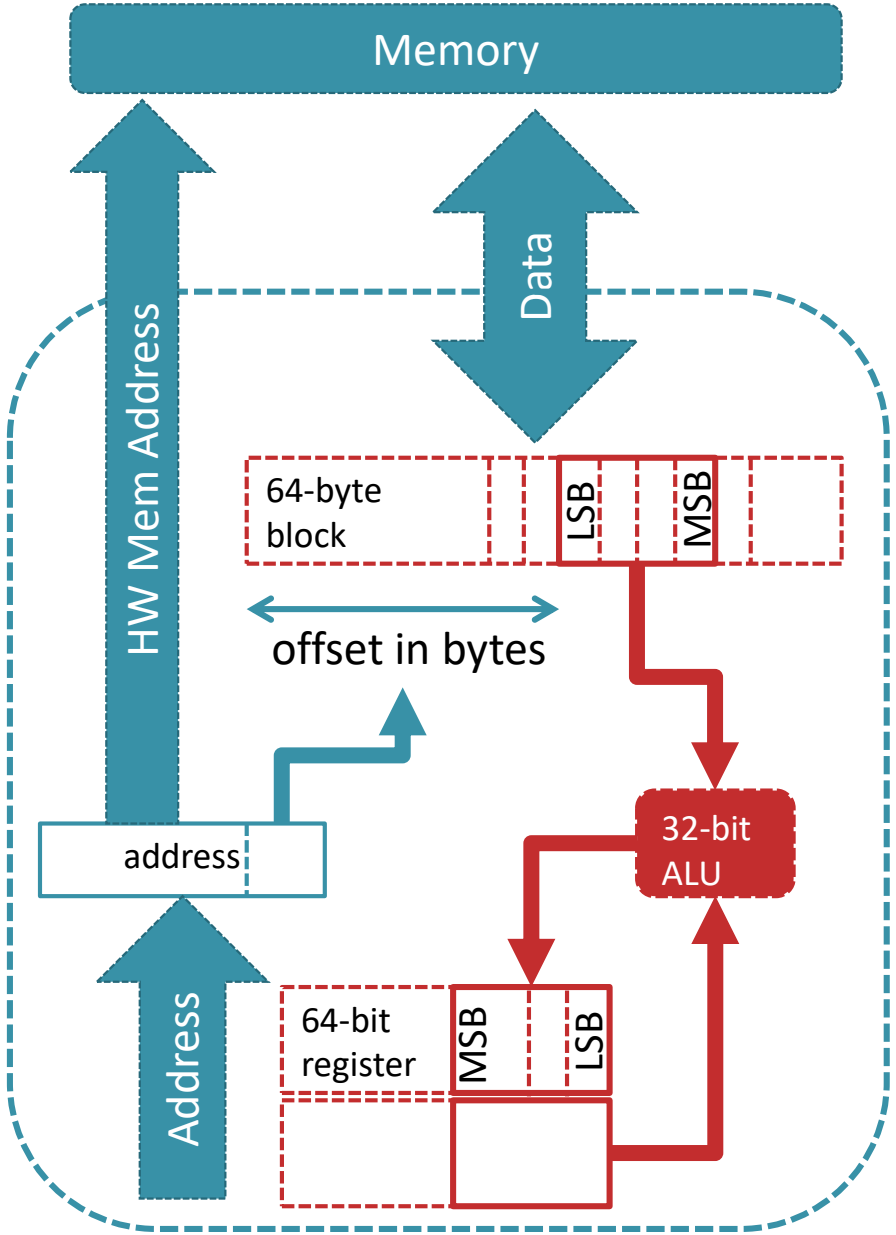
(část 2 - adresy)

Inside CPU – registers



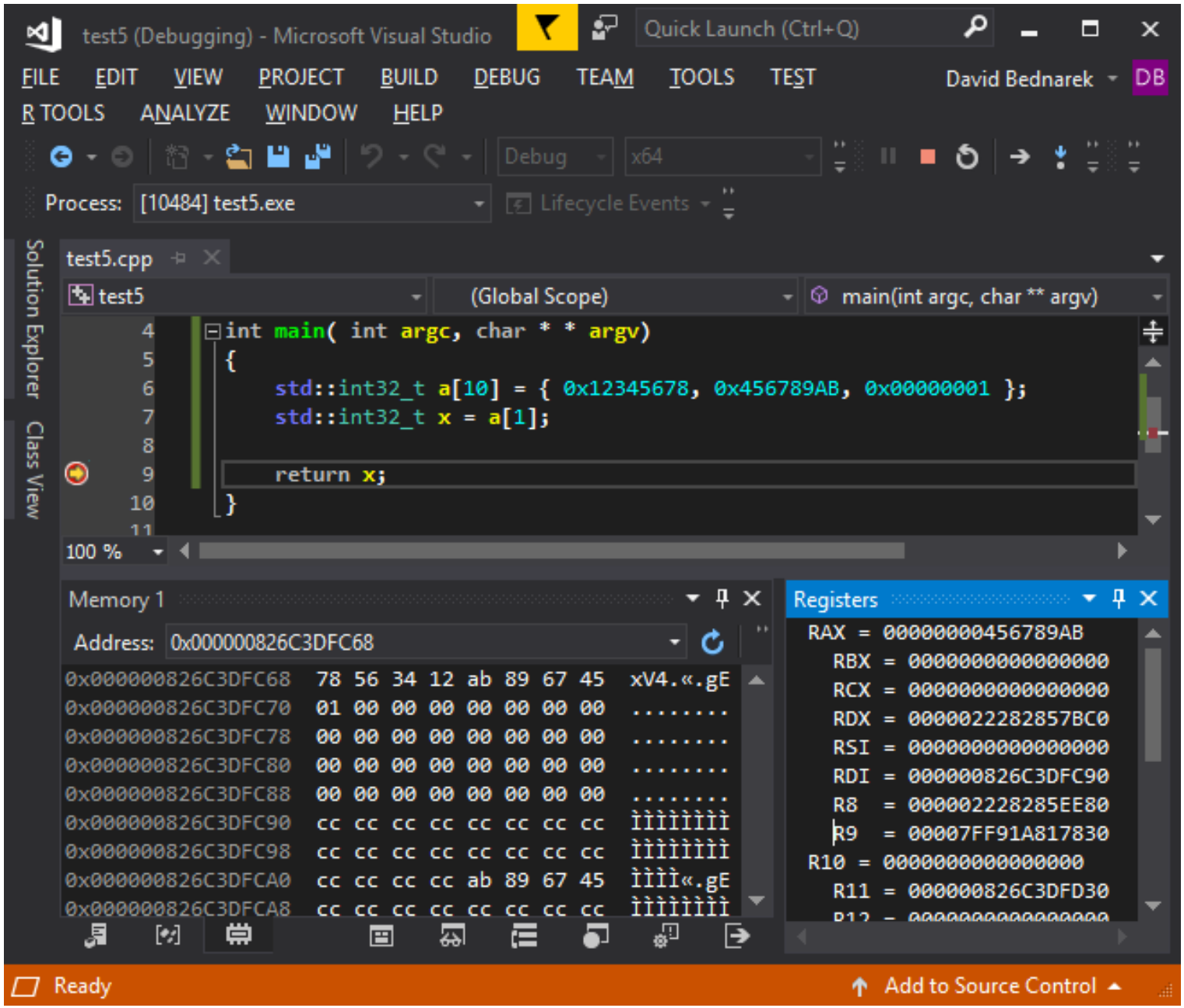
- Registers
  - Fastest storage available
    - register access: < 1 clock
    - memory (cache hit): ~ 4 clocks
    - memory (cache miss): ~ 120 clocks
  - Intel/AMD (64-bit):
    - 15 64-bit integer registers
      - 32/16/8-bit parts accessible
    - 8 80-bit FP registers
    - 16 256-bit vector registers [AVX2]
  - A typical instruction may access
    - 1 to 3 registers
    - at most 1 memory position
  - Names (numbers) of registers encoded in instruction code
    - No indirect access possible
  - Special registers
    - Instruction pointer [AMD64: RIP]
    - Stack pointer [AMD64: RSP]

Inside CPU – drawing conventions

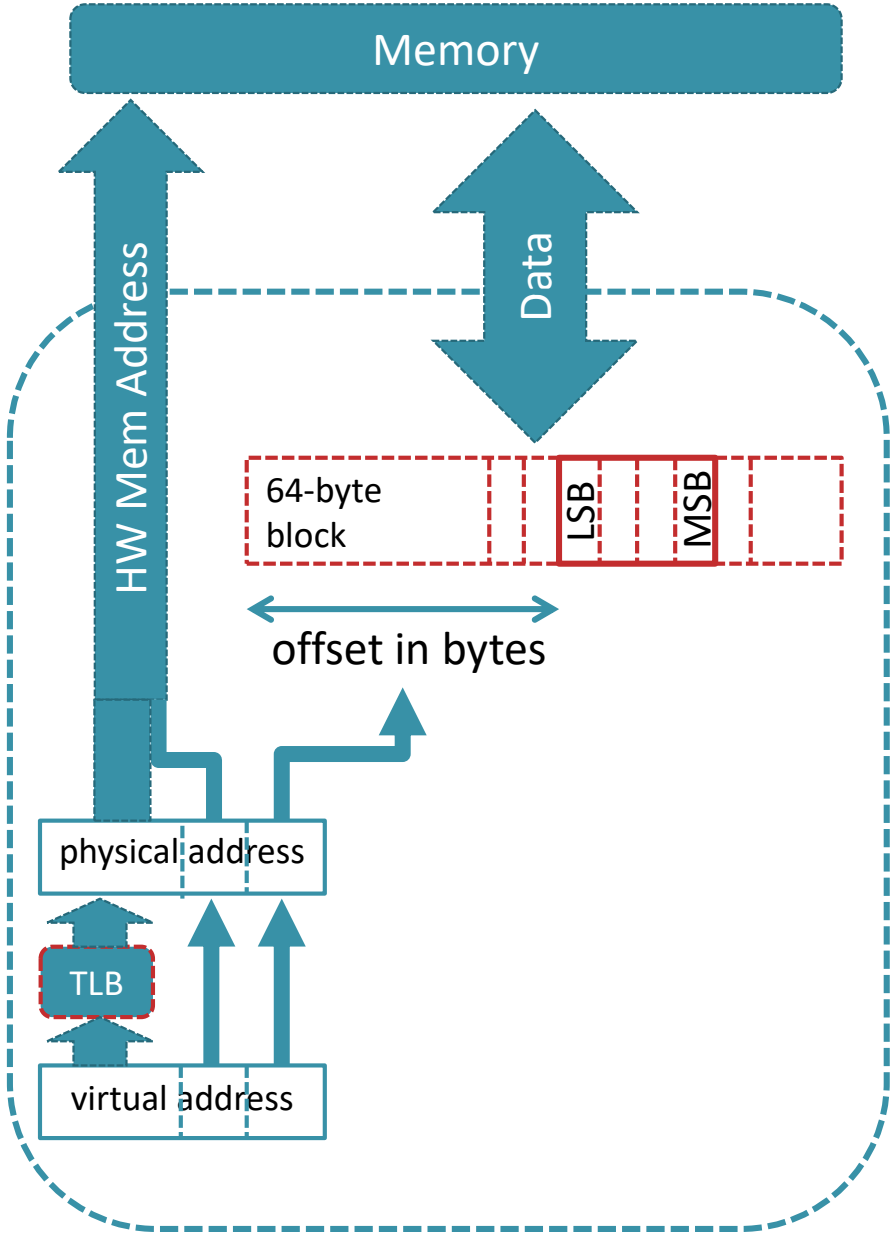


- Drawing conventions
  - Numbers are always written with most-significant-bit on the left
    - Convention since ~ 4th century BC (for Indian decimal numerals)
    - Read by humans starting with MSB - big-endian – consistent with left-to-right writing order
    - Registers are drawn in this order
  - Memory is usually drawn with lower addresses on the left
    - Derived from left-to-right writing order
  - In little-endian architectures, memory and register contents are *drawn* in opposite orders of bytes
    - 45 67 89 AB in register
    - AB 89 67 45 in memory
    - 0x456789AB in C/C++ code

Example – A Little-Endian CPU in a debugger

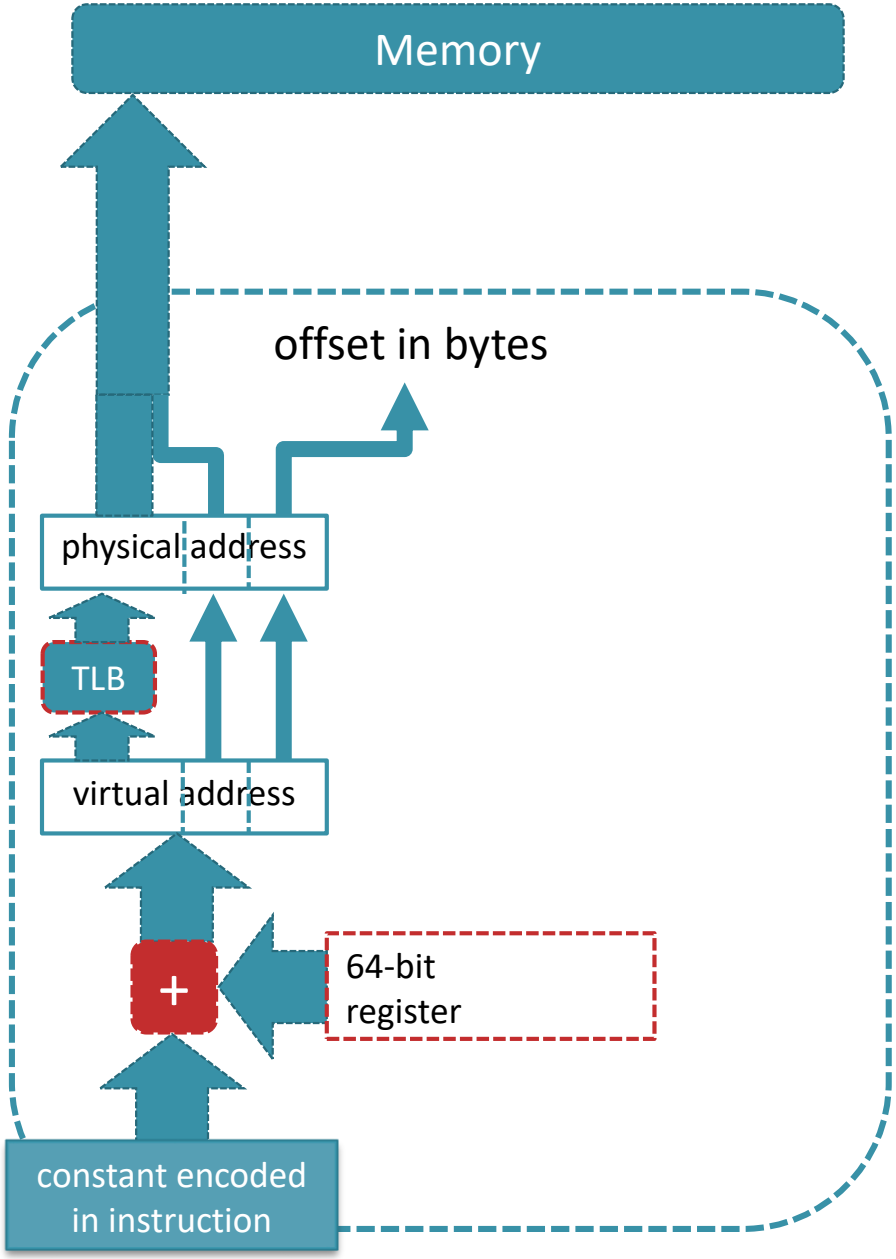


Inside CPU – virtual memory



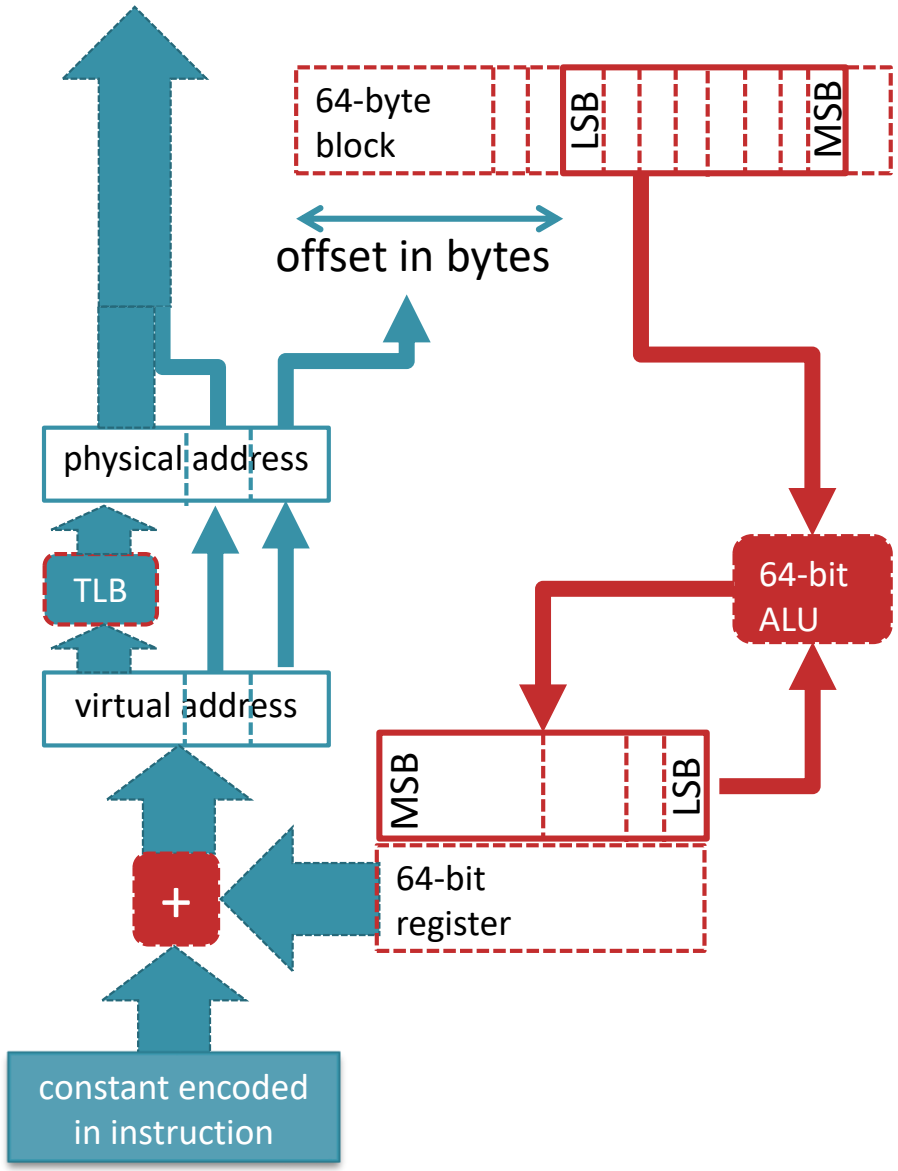
- TLB
  - Translation Look-aside Buffer
  - Translates upper part of addresses
  - Fast hardware mechanism
    - Associative memory
- Address not in TLB
    - or marked as protected
  - Either solved by *page-walk*
    - Slower hardware mechanism
  - Or causes *page-fault*
    - Invoke OS
- Page-fault
  - Solved by OS
  - either by *swapping*
    - Page data read from disk
  - or by *killing* the thread at fault

Inside CPU – where addresses come from



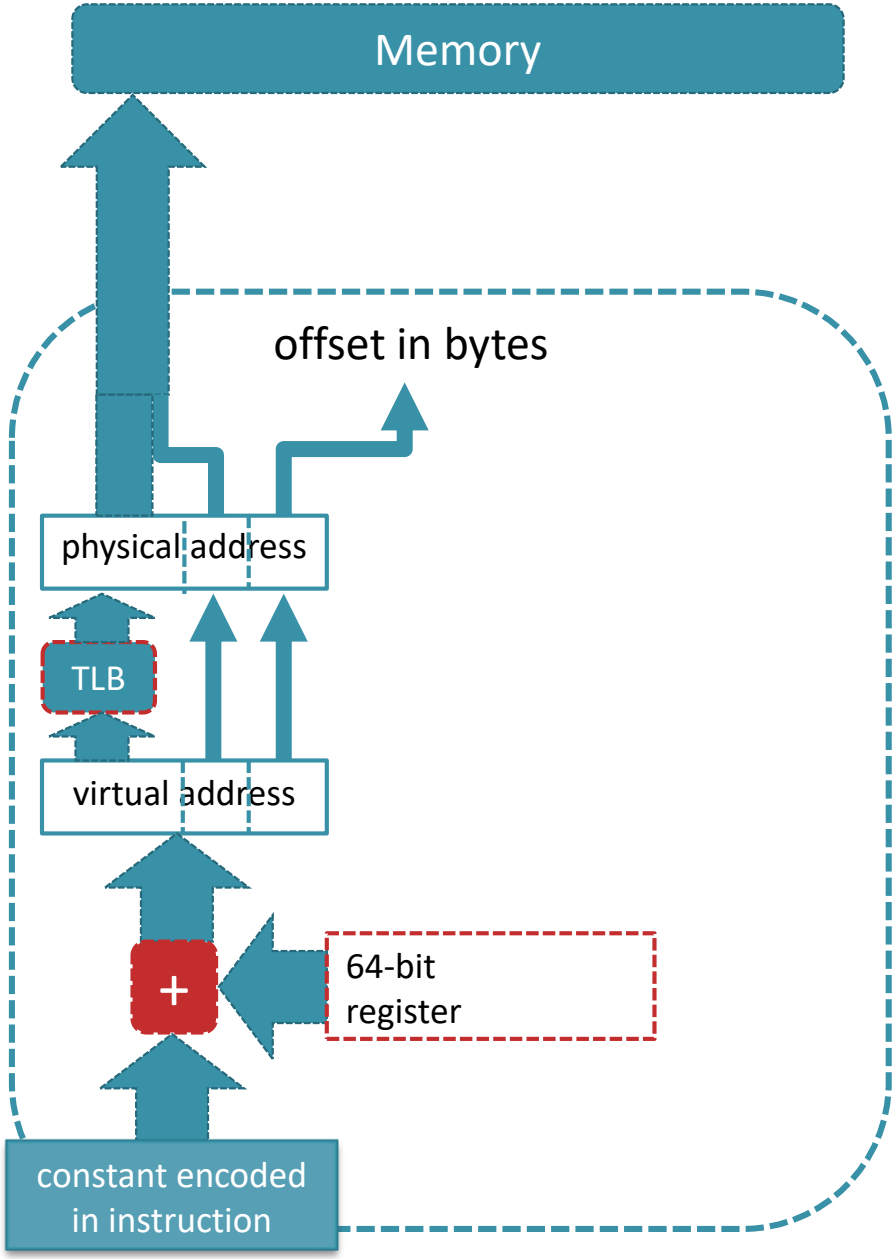
- Virtual address
  - 64-bit integer in most cases
    - Some upper bits ignored
  - 32-bit in 32-bit modes/CPUs
  - Generated as specified in the read/write instruction
  - Instruction encoding allows several *addressing modes*
  - Typical mode: register+constant
    - register name and constant value encoded in the instruction
    - constant value is computed by the compiler
      - may be adjusted by linker/loader
    - Some CPUs allow more complex modes
      - R1 + C1*R2 + C2

Inside CPU – summary of sizes



- Main memory and cache
  - Physically organized in 512-bit (64-byte) blocks
  - Blocks are invisible to software
    - But significantly affect performance
- Virtual addresses
  - An address denote an 8-bit area (byte) in memory
    - When accessing elementary data larger than 8 bits, the lowest address is specified
  - Computed using 64-bit arithmetics
    - This is why the CPU is called 64-bit
  - Upper 16 bits usually ignored
    - 256 TB virtual address space
- Physical addresses
  - Typically 35..42 bits (as of 2017)
    - 32 GB to 4 TB physical memory supported

Inside CPU – summary of addressing



- Addressing data in memory
  - The compiler constructs an expression which compute the *virtual address* of the data
  - In simple cases, the expression matches one of the addressing modes available
    - e.g. R1+C1
    - Complex cases may require additional instructions before the read/write
      - For CPU, address computation is just a 64-bit integer arithmetic
  - The address is translated by TLB
    - OS may be awaken to assist here
    - thread is killed if address is invalid
  - Physical address searched in cache
    - If not present, the block is read from main memory

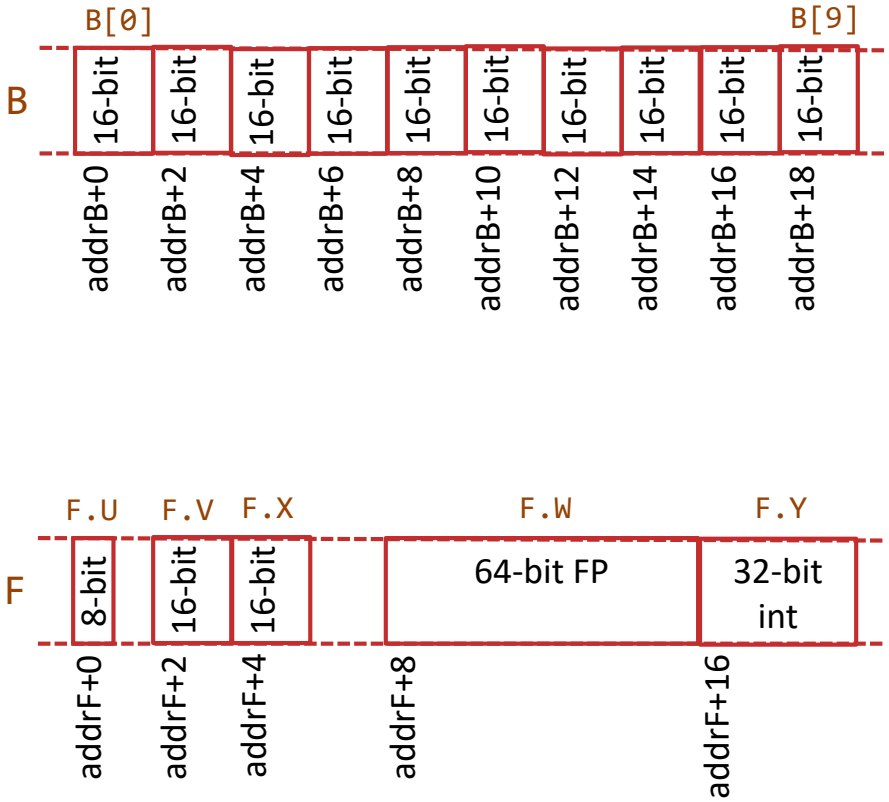
Storage classes

- Where the data reside?
  - Static storage
    - Global, static member, static local variables, string constants
      - One instance (per process)
    - Allocated by compiler/linker/loader (represented in .obj/.dll/.exe files)
  - Thread-local storage

C++11
    - Variables marked "thread_local"
      - One instance per thread
  - Automatic storage (stack or register)
    - Local variables, parameters, anonymous objects, temporaries
      - One instance per invocation of the enclosing function (pass through the declaration)
    - Placement planned by the compiler, allocation done by instructions generated by the compiler, usually once for all variables in a function
  - Dynamic allocation
    - new/delete operators
      - Programmer is responsible for allocation/deallocation – no garbage collection exists in C++
    - new/delete translates to library function calls
      - Significantly slower allocation than automatic storage

C++11
    - Use smart pointers (built atop new/delete) instead
      - Allocation by library functions, deallocation when the last smart pointer disappears

Data in memory – arrays and structures as static variables



- Arrays
  - If B is static (global) variable

```
static std::int16_t B[10];
```

    - Address is assigned by compiler

```
x = B[I];
```

    - Translated to something like

```
mov tmp,I
shl tmp,1              ; 64-bit multiply by 2
mov x,[addrB+tmp]      ; 16-bit memory read
```

    - 64-bit address computed at runtime

- Structures
  - If F is static (global) variable

```
struct S { /*...*/ }; static S F;
y = F.V;
```

    - Translated to something like

```
mov y,[addrF+2]        ; 16-bit memory read
```

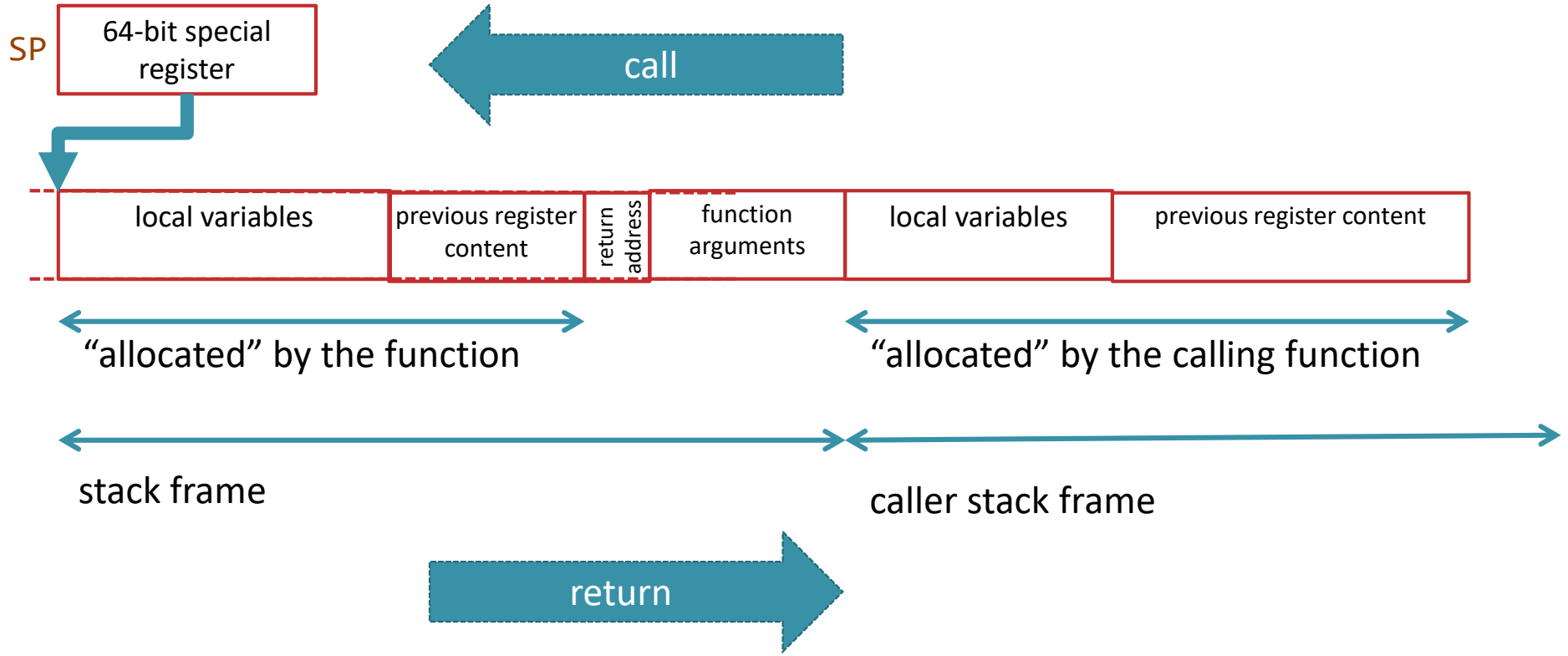
    - address computed by compiler

Data in memory – automatic storage

- Automatic storage
  - Variables declared inside a function (except when marked static)
- Formally
  - Variable is created when control passes through its declaration
  - Variable is destroyed when the enclosing compound statement is exited
- In typical implementation
  - The space for the variable is reserved when entering the function
    - The space may be reused for other variables if not used at the same moment
  - Constructor (if any) is called when control passes the declaration
  - Destructor is called when exiting the compound statement

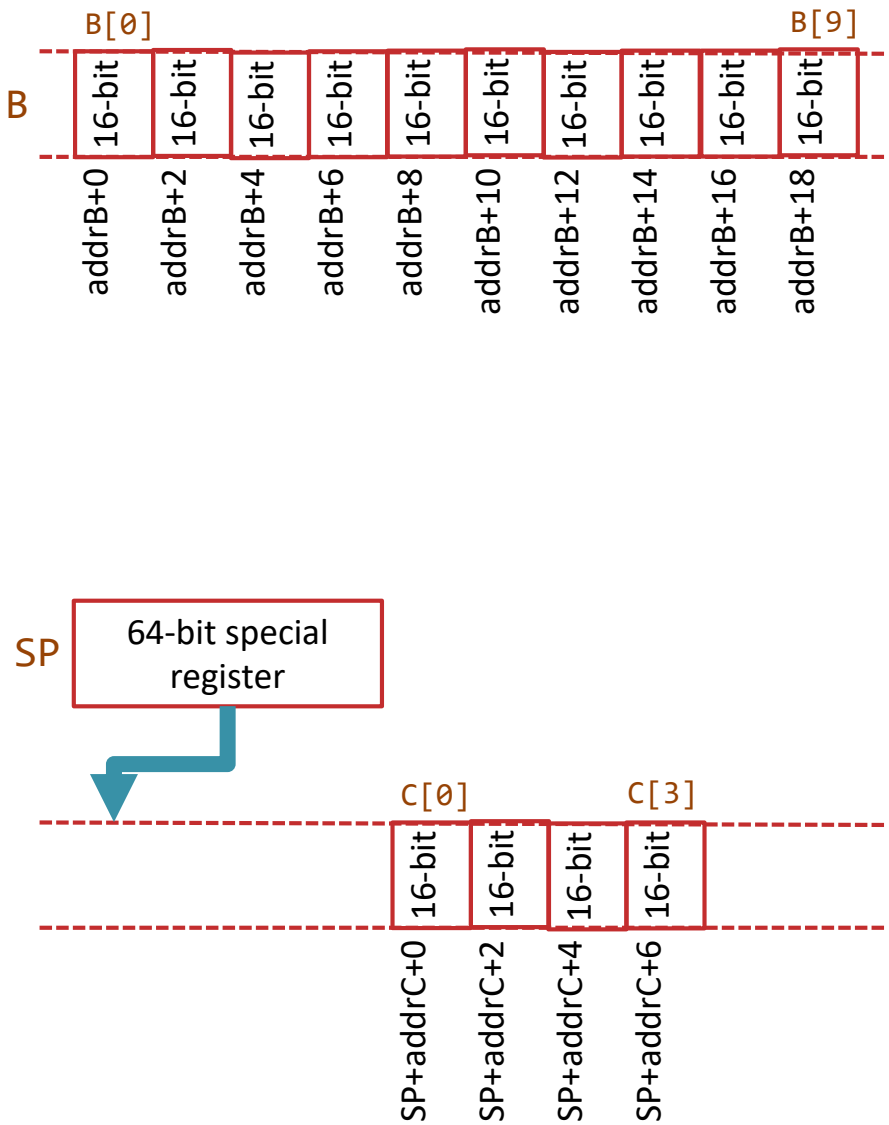
- Scalar variables (elementary types, decomposed structures)
  - Preferably in registers
    - Previous contents of registers must be saved to the stack when entering the function and restored on exit
  - The rest in stack
- Aggregate variables (arrays, non-decomposed structures)
  - Must be located in stack (registers do not support addressing/indexing)
- The compiler plans register numbers and relative stack positions
  - Registers are just used (after saving previous values)
  - Stack is (de)allocated by adding/subtracting a constant from the SP register
    - There may be run-time checks for overflowing the total stack space

Data in memory – stack frames



- Stack frame
  - local variables, arguments and other info on the stack
  - layout planned by the compiler
  - can be inspected by debuggers or crash dump analyzers
- This picture assumes stack growing towards lower addresses (as in Intel/AMD)

Data in memory – statically vs. automatically allocated arrays



- *Static storage* (static/global) variable

```
static std::int16_t B[10];
```

  - Variable resides at fixed place (*addrB*)

```
x = B[I];
```

  - read I
  - multiply by element size
  - add constant *addrB*
  - read from memory
  - write to x

- *Automatic storage* (local) variable

```
void f() {
  std::int16_t C[4];
```

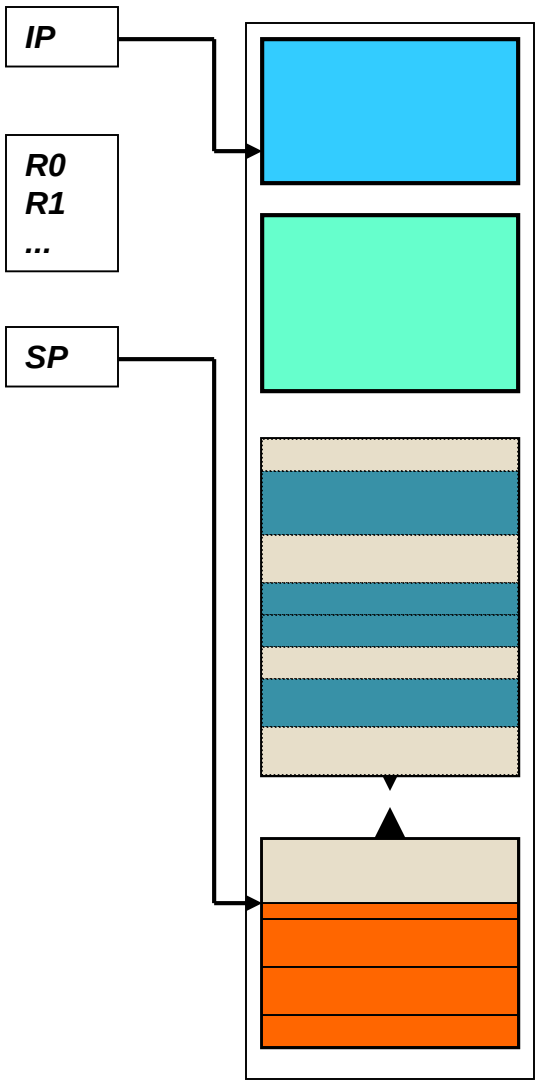
  - The size of array must be determined by the compiler
  - The variable is *the array*

```
  x = C[I];
```

  - read I
  - multiply by element size
  - add SP
  - add constant addrC
  - read from memory
  - write to x

```
}
```

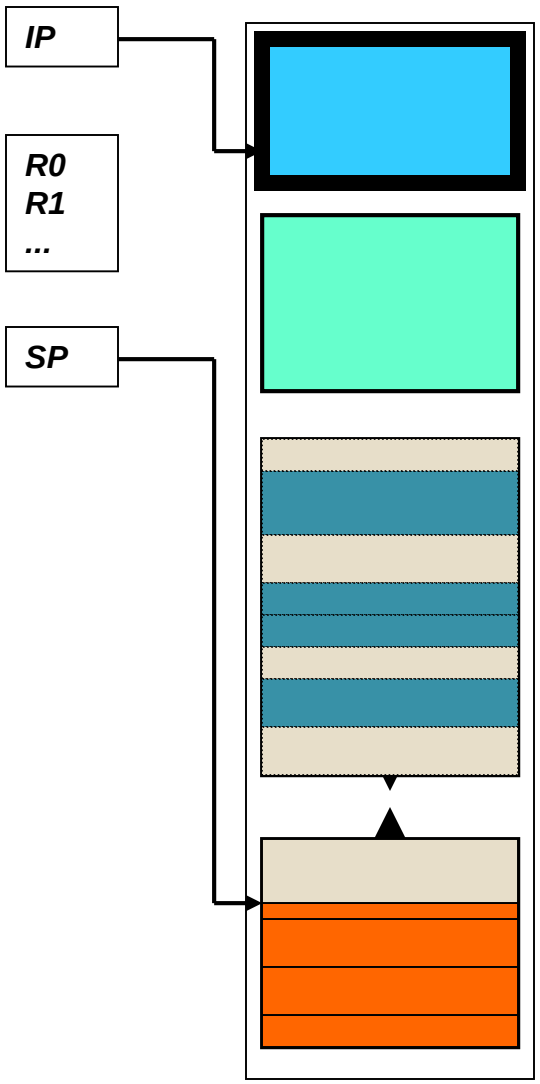
Organizace paměti procesu



- Kódový segment
- Datový segment
- Heap
- Zásobník (stack segment)

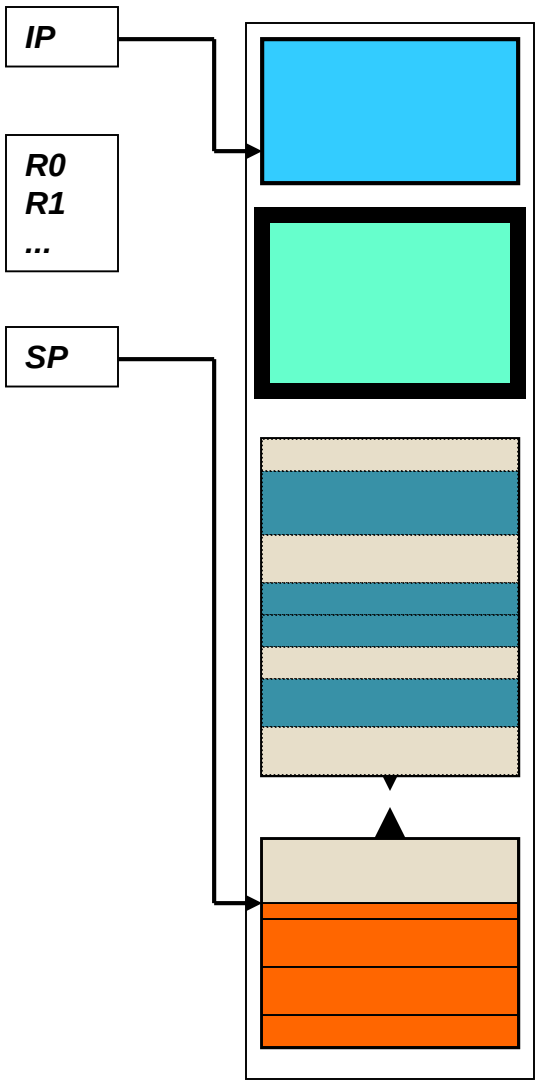
  - Segmenty (vyjma zásobníku) nemusejí být souvislé
    - Dynamicky-linkované knihovny sdílené mezi procesy
    - Postupná alokace heapu

Organizace paměti procesu



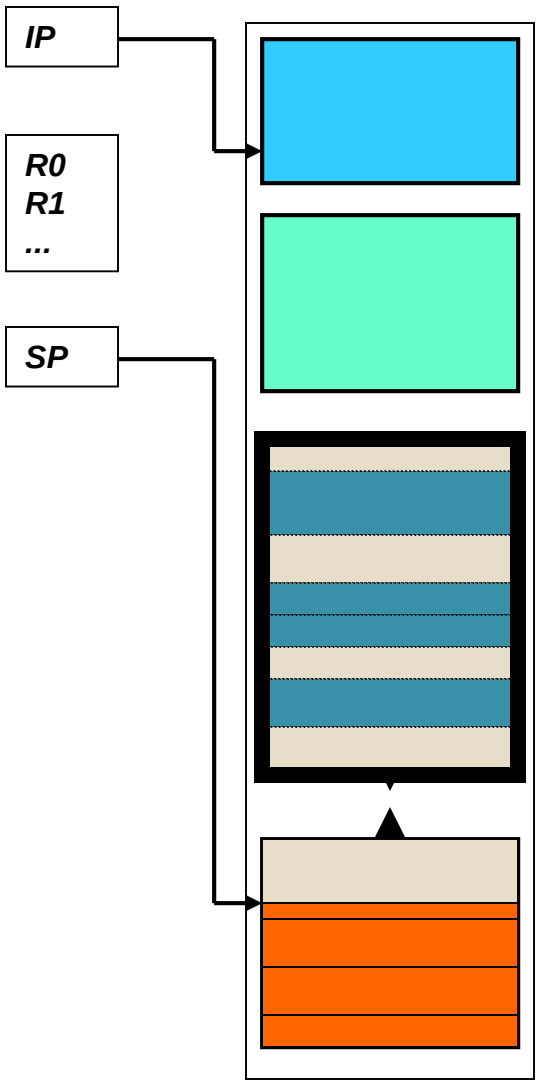
- Kódový segment
  - Připraven kompilátorem – součást spustitelného souboru
    - Kód uživatelských i knihovních funkcí
    - Obvykle chráněn proti zápisu
- Datový segment
- Heap
- Zásobník (stack segment)

Organizace paměti procesu



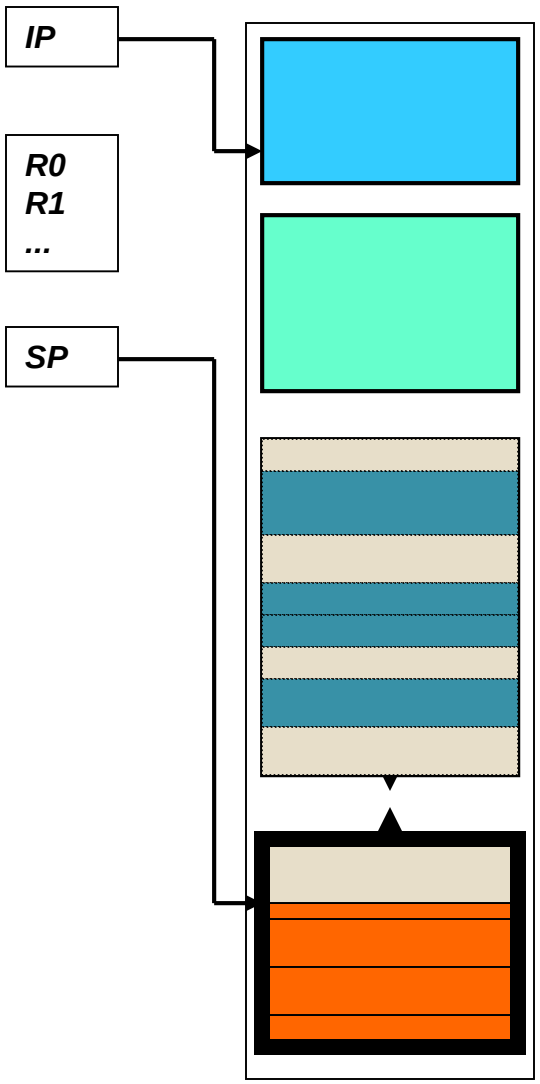
- Kódový segment
- Datový segment
  - Připraven kompilátorem – součást spustitelného souboru
    - Explicitně nebo implicitně (nulami) inicializované globální proměnné
    - Řetězcové konstanty
    - Data knihoven
    - Pomocná data generovaná kompilátorem (vtables apod.)
- Heap
- Zásobník (stack segment)

Organizace paměti procesu



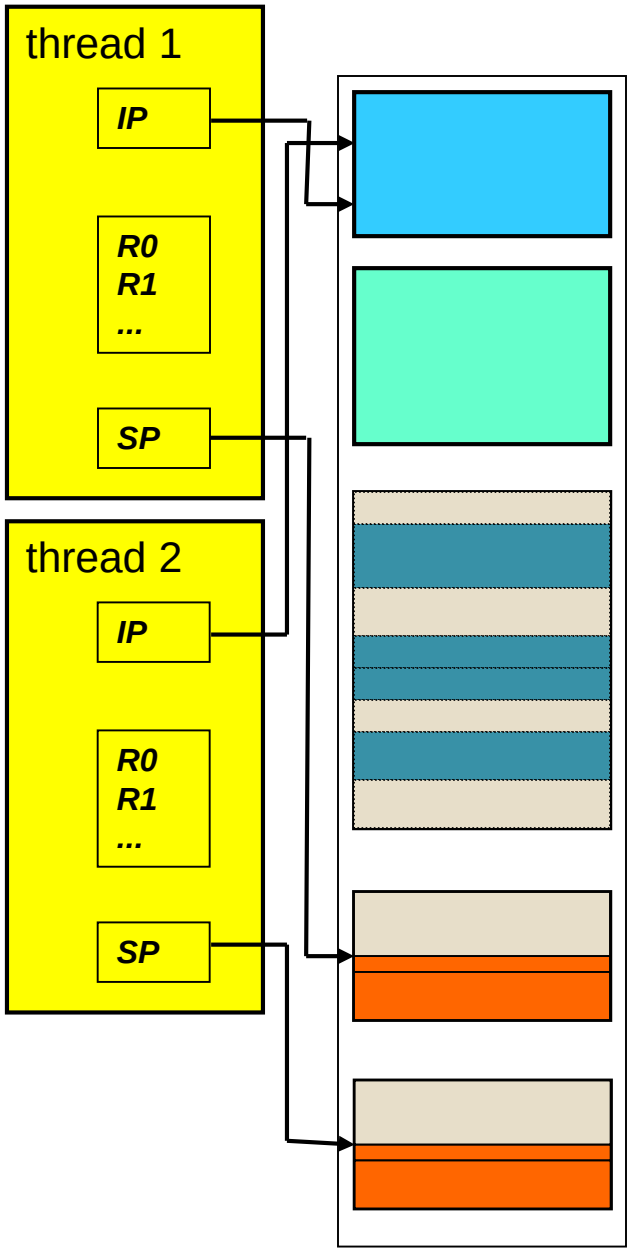
- Kódový segment
- Datový segment
- Heap
  - Vytvářen startovacím modulem knihoven
    - Neinicializovaná dynamicky alokovaná data
    - C++: new/delete
    - C: malloc/free
    - Obsazené bloky různé velikosti + seznam volných bloků
    - Knihovny mohou též požádat OS o zvětšení segmentu
- Zásobník (stack segment)

Organizace paměti procesu



- Kódový segment
- Datový segment
- Heap
- Zásobník (stack segment)
  - Připraven op. systémem, knihovny mohou požádat OS o zvětšení
    - Explicitně inicializované nebo neinicializované lokální proměnné
    - Pomocné proměnné generované kompilátorem
    - Návratové adresy
    - Další pomocná data
  - Vícevláknové aplikace mají více zásobníků

Organizace paměti vícevláknového procesu



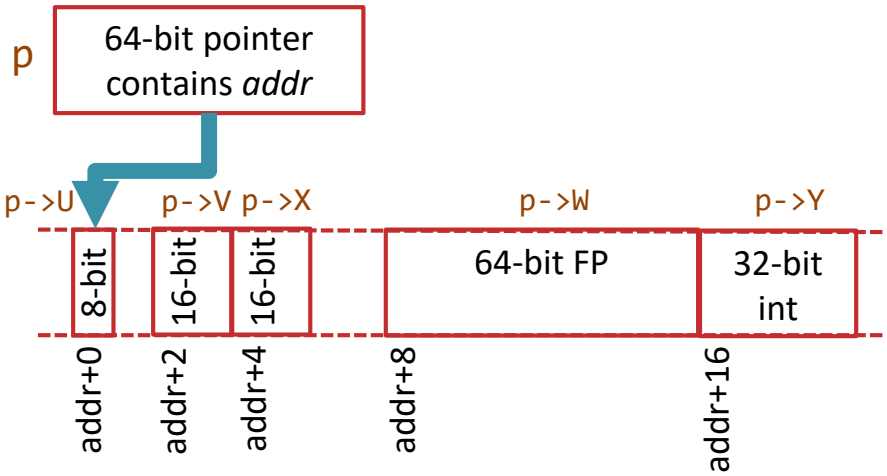
- Vlákno z pohledu OS
  - IP – Ukazatel instrukcí
  - SP – Ukazatel zásobníku
  - Další registry procesoru
  - (Identifikátor vlákna)
- Paměťový prostor je společný
- Vlákno v paměťovém prostoru
  - Zásobník
  - Thread-local storage
    - Na dně zásobníku, nebo
    - lokalizováno dle id vlákna

Dynamická alokace

- Dynamická alokace je pomalá
  - Ve srovnání s automatickou
  - Důvodem je především chování cache, v menší míře samotný algoritmus alokace
- Užívejte dynamickou alokaci, jen když je to nutné
  - velká pole a/nebo pole s proměnlivou velikostí
  - polymorfní kontejnery (pro objekty s dědičností)
  - životnost objektu nesouhlasí s vyvoláním nějaké funkce v programu
- Vyhýbejte se datovým strukturám s individuálně alokovanými prvky
  - spojové seznamy, binární stromy, ...
    - std::list, std::map, ...
  - raději B-stromy (i v paměti) nebo hašování
  - vyhýbání se je obtížné – má smysl, jen pokud opravdu na rychlosti záleží

- Takto lze udělat programy v C++ znatelně rychlejší než v C#/javě
  - C#/java povinně alokuje každý objekt dynamicky
- Když budete v C++ programovat stylem převzatým z C#/javy, budou vaše programy skoro stejně pomalé

Data in memory – dynamically allocated structures in C++



- Structure allocated dynamically

```
struct S { /*...*/ };

S * p = new S;
```

  - S * = (raw) pointer to S
    - may be replaced by auto
  - new: (Raw) dynamic allocation
    - library function call (+ constructor call)
  - variable p itself is in automatic storage (register)
  - variable p contains *addr* returned from *new*
    - an address of a 24-byte block
    - aligned to 8-byte boundary

```
y = p->V;        // same as y = (*p).V
```

  - Translated to something like

```
mov tmp,p

mov y,[2+tmp]    ; 16-bit memory read
```

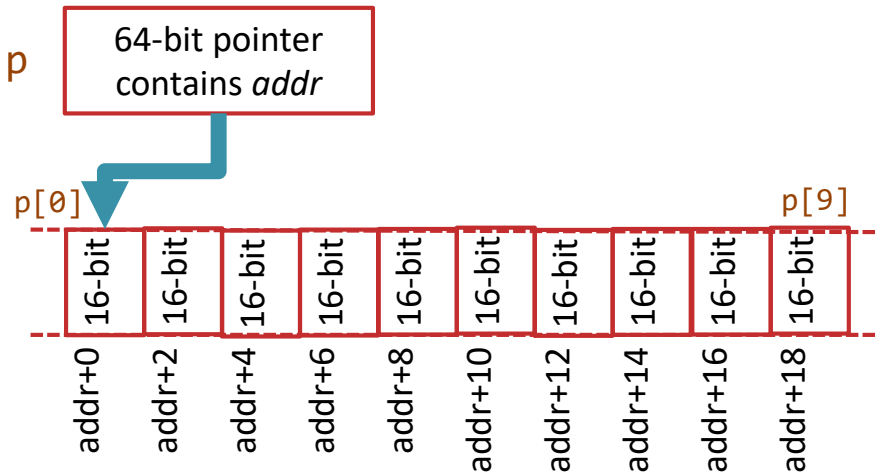
  - note: this is the same memory-read instruction as when reading static array
    - addressing mode: constant + register

- There is no garbage collector
  - The block shall be explicitly freed

```
delete p;

p = nullptr;
```

  - delete: (destructor call +) library function call
    - beware: delete does not alter the pointer itself
  - pointer explicitly nulled (safety first)
- Using raw pointers and new/delete is not recommended in C++11

Data in memory – dynamically allocated arrays in C++



- Array allocated dynamically

```
std::size_t N = 10;
```

  - The size of array may be determined at run-time

```
std::int16_t * p = new std::int16_t[N];
```

  - std::int16_t * = (raw) pointer to std::int16_t
    - may be replaced by auto
  - Arrays are held by pointers to the first element
    - Convention supported by language rules
    - Beware: The size of the array is not a part of p
  - Variable p contains *addr* returned from *new*
    - an address of a (2*N)-byte block
    - aligned to 2-byte boundary

```
x = p[I];
```

  - The same syntax for pointers and arrays
    - The action is slightly different
  - Translated to something like

```
mov tmp,I
shl tmp,1                    ; 64-bit shift left
add tmp,p                    ; 64-bit addition
mov x,[tmp]                  ; 16-bit memory read
```

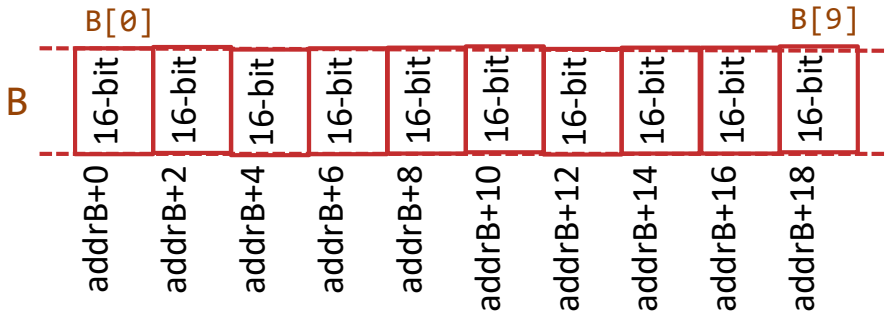
    - 64-bit address computed at runtime

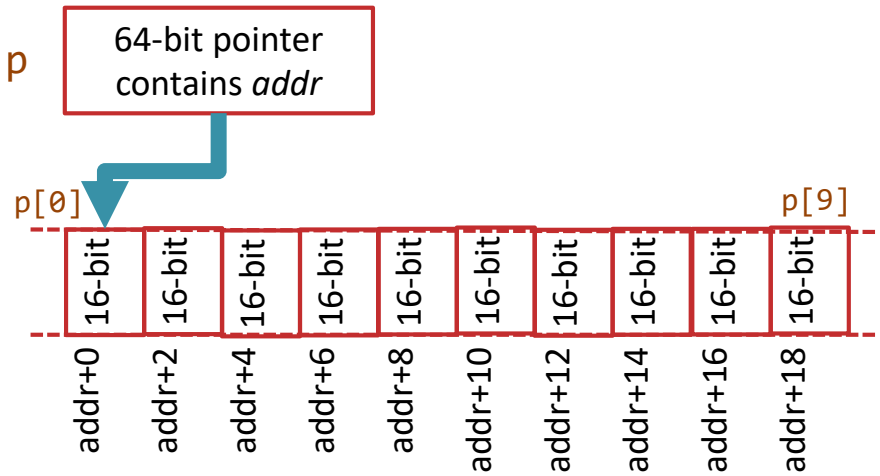
  - The block shall be explicitly freed

```
delete[] p;
p = nullptr;
```

    - The runtime is able to determine the size of the block

Data in memory – statically vs. dynamically allocated arrays





- Statically allocated (static/global) variable of array type

```
static std::int16_t B[10];
```

  - The size of array must be determined by the compiler
  - The variable is *the array*

```
x = B[I];
```

  - read I
  - multiply by element size
  - add constant *addrB*
  - read from memory
  - write to x

- Dynamically allocated array

```
std::int16_t * p = new std::int16_t[N];
```

  - The size of array may be determined at run-time
  - The variable is *a pointer* to the array

```
x = p[I];
```

  - read I
  - multiply by element size
  - read p
  - add
  - read from memory
  - write to x

Storage classes

- Kde jsou umístěna data...
  - Static storage (datový segment)

```
T x;      // global variable
```

  - Thread-local storage

```
thread_local T x;  // global variable
```

  - Automatic storage (zásobník nebo registr)

```
void f() {
  T x;    // local variable
}
```

  - Dynamic allocation (heap) – v C++11 zastaralý styl:

```
void f() {
  T * p = new T;
  // ...
  delete p;
}
```

Vracení odkazem a hodnotou

Vracení odkazem

- Funkce jako add nemůže vracet referenci
  - add vrací hodnotu různou od všech svých parametrů
  - hodnotu parametrů nesmí měnit
  - reference nemá na co ukazovat

  - Špatné řešení č. 1: Lokální proměnná

```
Complex & add( const Complex & a, const Complex & b)
{
  Complex r( a.Re + b.Re, a.Im + b.Im);
  return r;
}
```

  - BĚHOVÁ CHYBA: r zaniká při návratu z funkce

Vracení odkazem

- Funkce jako add nemůže vracet referenci
  - add vrací hodnotu různou od všech svých parametrů
  - hodnotu parametrů nesmí měnit
  - reference nemá na co ukazovat

  - Špatné řešení č. 2: Dynamická alokace

```
Complex & add( const Complex & a, const Complex & b)
{
  Complex * r = new Complex( a.Re + b.Re, a.Im + b.Im);
  return * r;
}
```

  - PROBLÉM: kdo to odalokuje ?

Vracení odkazem

- Funkce jako add nemůže vracet referenci
  - add vrací hodnotu různou od všech svých parametrů
  - hodnotu parametrů nesmí měnit
  - reference nemá na co ukazovat

  - Špatné řešení č. 3: Globální proměnná

```
Complex g;
Complex & add( const Complex & a, const Complex & b)
{
  g = Complex( a.Re + b.Re, a.Im + b.Im);
  return g;
}
```

  - CHYBA: globální proměnná je sdílená

```
Complex a, b, c, d, e = add( add( a, b), add( c, d));
```

Vracení odkazem

- Funkce jako add musí vracet hodnotou
  - add vrací hodnotu různou od všech svých parametrů
  - hodnotu parametrů nesmí měnit
  - reference nemá na co ukazovat

  - Správné řešení

```
Complex add( const Complex & a, const Complex & b)
{
  Complex r( a.Re + b.Re, a.Im + b.Im);
  return r;
}
```

  - Zkrácený (ekvivalentní) zápis

```
  return Complex( a.Re + b.Re, a.Im + b.Im);
```

Vracení referencí

- Funkce které *umožňují přístup* k existujícím objektům mohou vracet *referenci*
  - Objekt musí přežít i po návratu z funkce
  - Příklad:

```
template< typename T, std::size_t N> class array {
public:
  T & at( std::size_t i)
  {
    return a_[ i];
  }
private:
  T a_[ N];
};
```

  - Vracení reference může umožnit i modifikaci objektu

```
array< int, 5> x;
x.at( 1) = 2;
```

Hello, World!

Hello, World!

```
#include <iostream>


int main( int argc, char * * argv)
{
  std::cout
    << "Hello, world!"
    << std::endl;

  return 0;
}
```

- Vstupní bod programu
  - Dědictví jazyka C
    - Žádné třídy ani metody
  - Globální funkce main
- Parametry programu
  - Z příkazové řádky
    - Děleno na kousky
  - Archaické datové typy
    - Ukazatel na ukazatel
    - Logicky pole polí
- std - namespace knihoven
- cout - standardní výstup
  - globální proměnná
- << - výstup do streamu
  - přetížený operátor
- endl - oddělovač řádek
  - globální funkce (!)

Hello, World!

Dělení do modulů

- Rozhraní modulů je nutno opsat do zvláštního souboru
  - .hpp - hlavičkový soubor
- Definující i používající modul tento soubor inkluduje
  - textová direktiva #include

```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

void world();

#endif
```

```
// main.cpp
#include "world.hpp"


int main( int argc, char * * argv)
{
  world();
  return 0;
}
```

```
// world.cpp
#include "world.hpp"
#include <iostream>


void world()
{
  std::cout << "Hello, world!"
      << std::endl;
}
```

Hello, World!

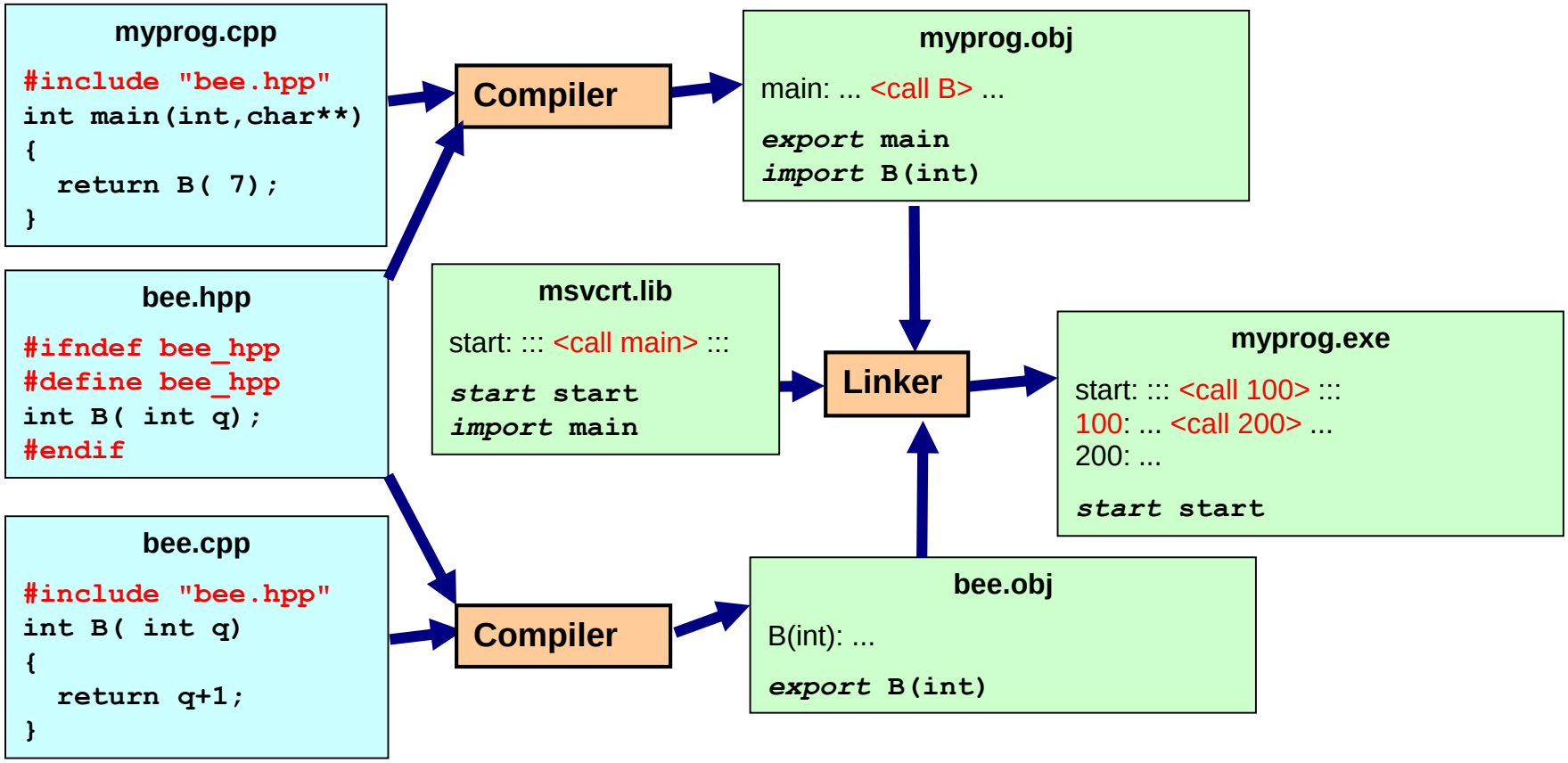
```cpp
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

using t_arg = std::vector< std::string>;
void world( const t_arg & arg);

#endif
```

```cpp
// main.cpp
#include "world.hpp"

int main( int argc, char * * argv)
{
  world( t_arg( argv + 1, argv + argc));
  return 0;
}
```

```cpp
// world.cpp
#include "world.hpp"
#include <iostream>

void world( const t_arg & arg)
{
  if ( arg.empty() )
  {
    std::cout << "Hello, world!"
      << std::endl;
  }
}
```

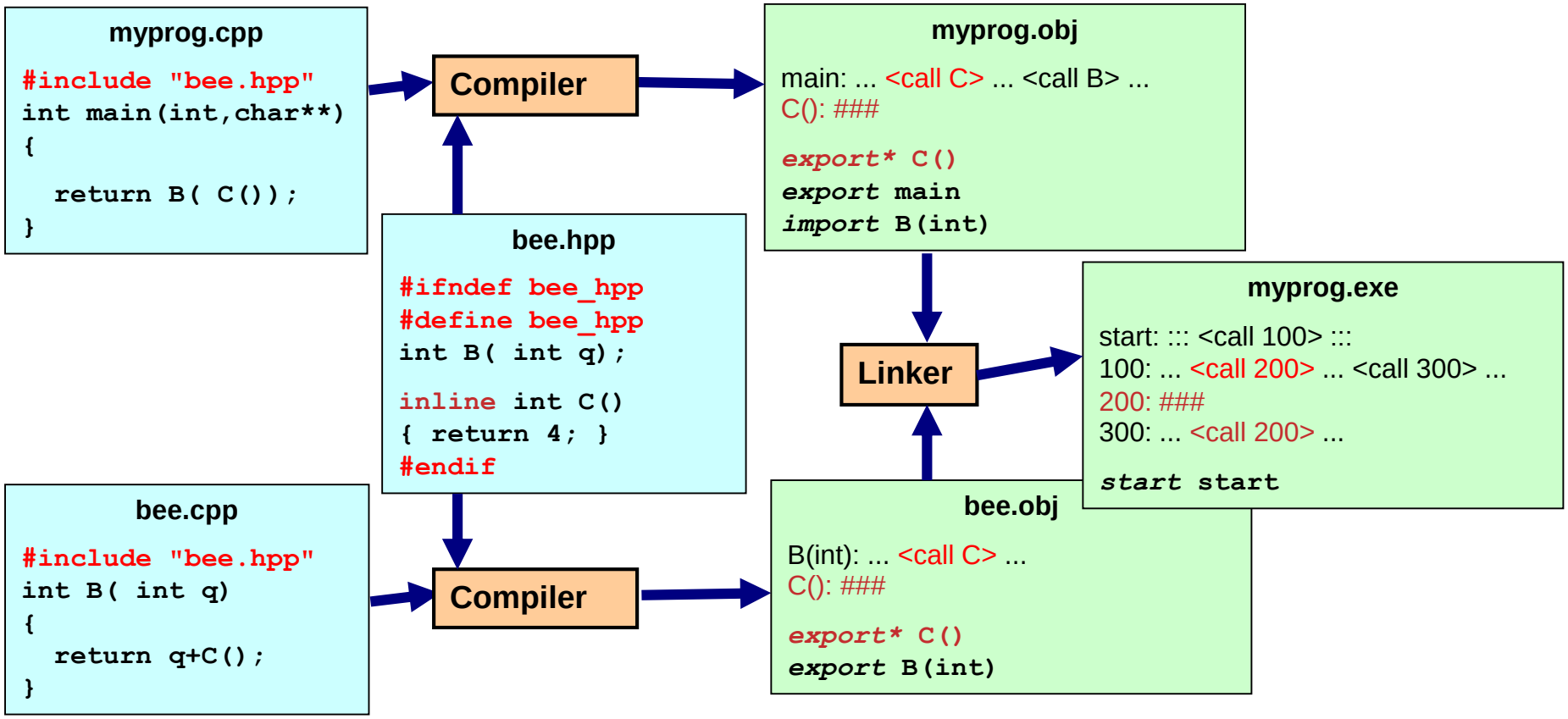
Compilation and linking

Independent compilation of modules (old style)



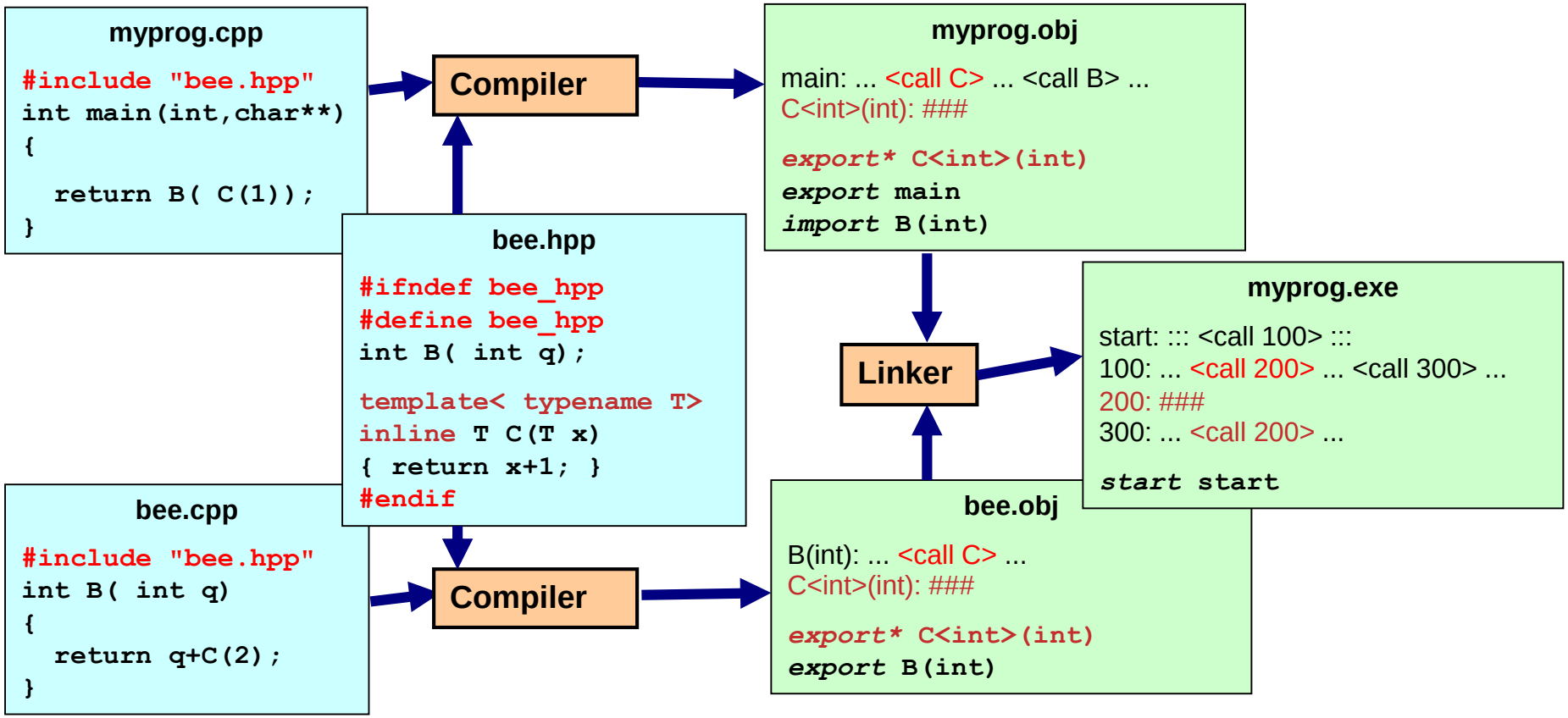
- Non-inline function B defined in bee.cpp, called from myprog.cpp
  - Declaration of B shared in bee.hpp
- Body of B is compiled and optimized only once
- Old style compilers produce binary code of target CPU

Inline functions (old style)



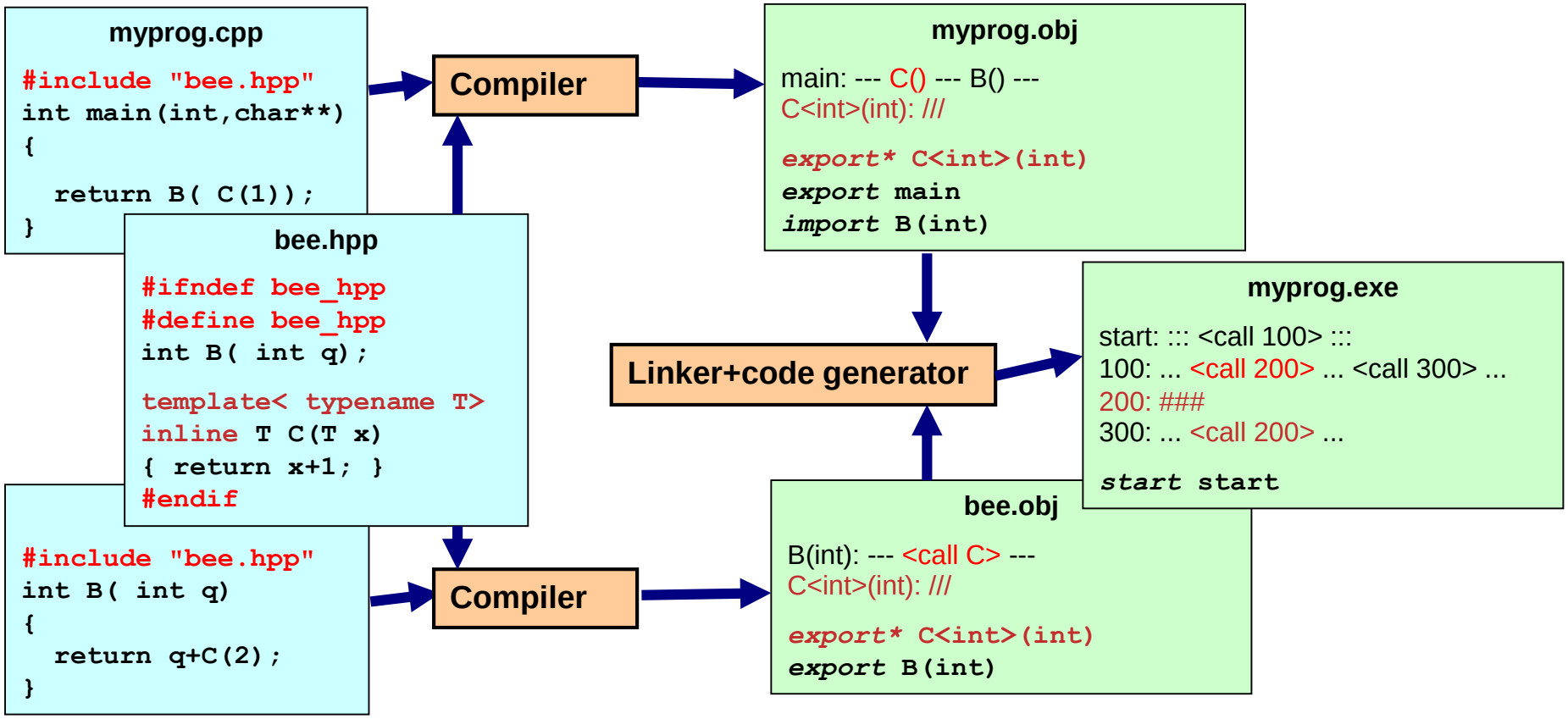
- Inline function C defined in bee.hpp, called from myprog.cpp and bee.cpp
- Body of C is compiled and optimized twice
  - The compiler *may* place the function body instead of the call (inlining aka procedure integration)
    - The compiler may do inlining even if the function is not marked as inline
  - If not inlined, the inline keyword ensures that linker ignores duplicates
    - Function bodies inside classes/structs are automatically considered inline

Template inline functions (old style)



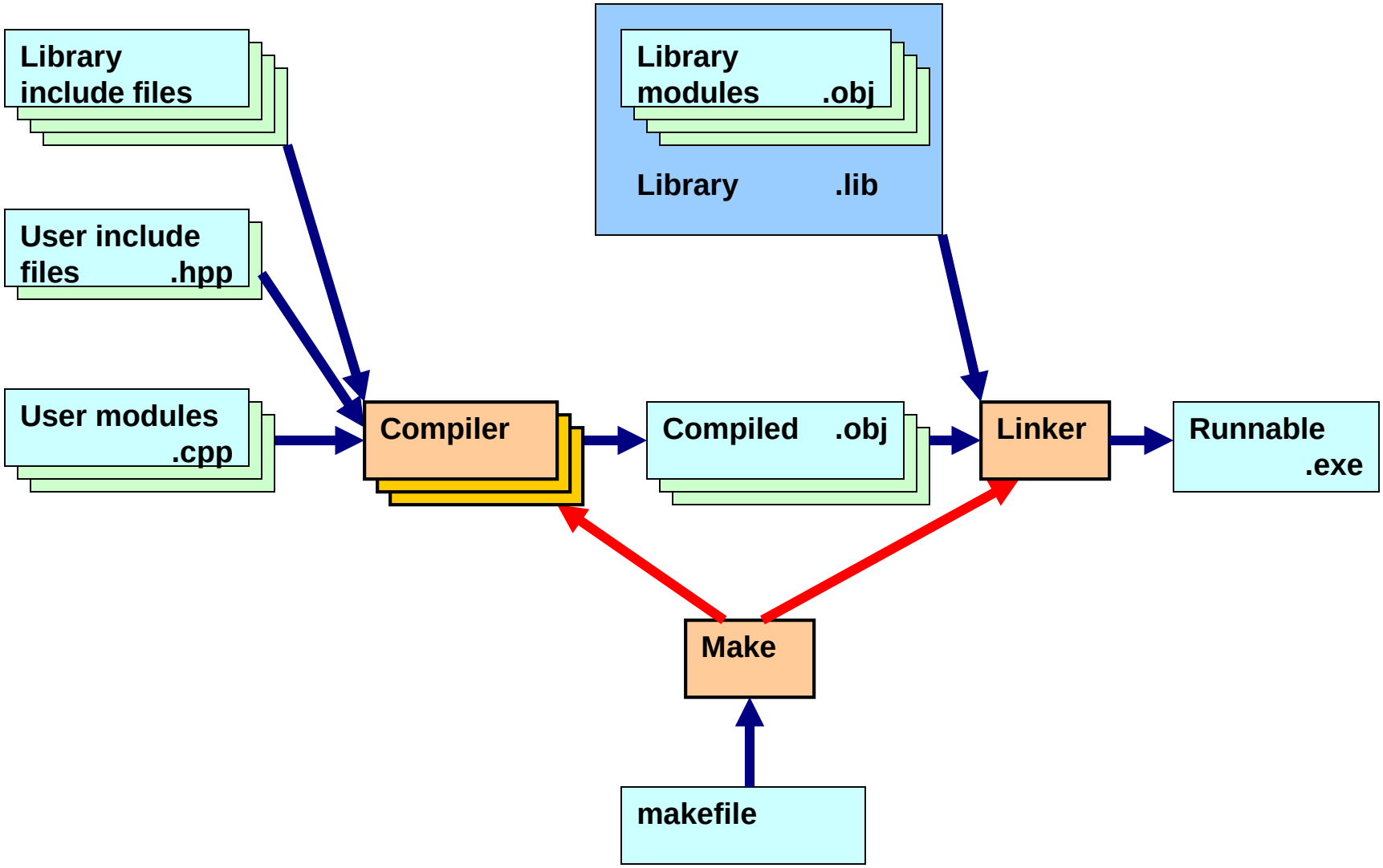
- Template inline function C defined in bee.hpp, called from myprog.cpp and bee.cpp
- Body of C is instantiated as C<int>, compiled and optimized twice
- The body of C must be visible for the compiler which does the instantiation
  - Otherwise, there will be no compiler to instantiate it
- All template code must be in a header file (and therefore inline)
  - Except for module-local templates or special-case tricks with explicit instantiation

Template inline functions (modern style)

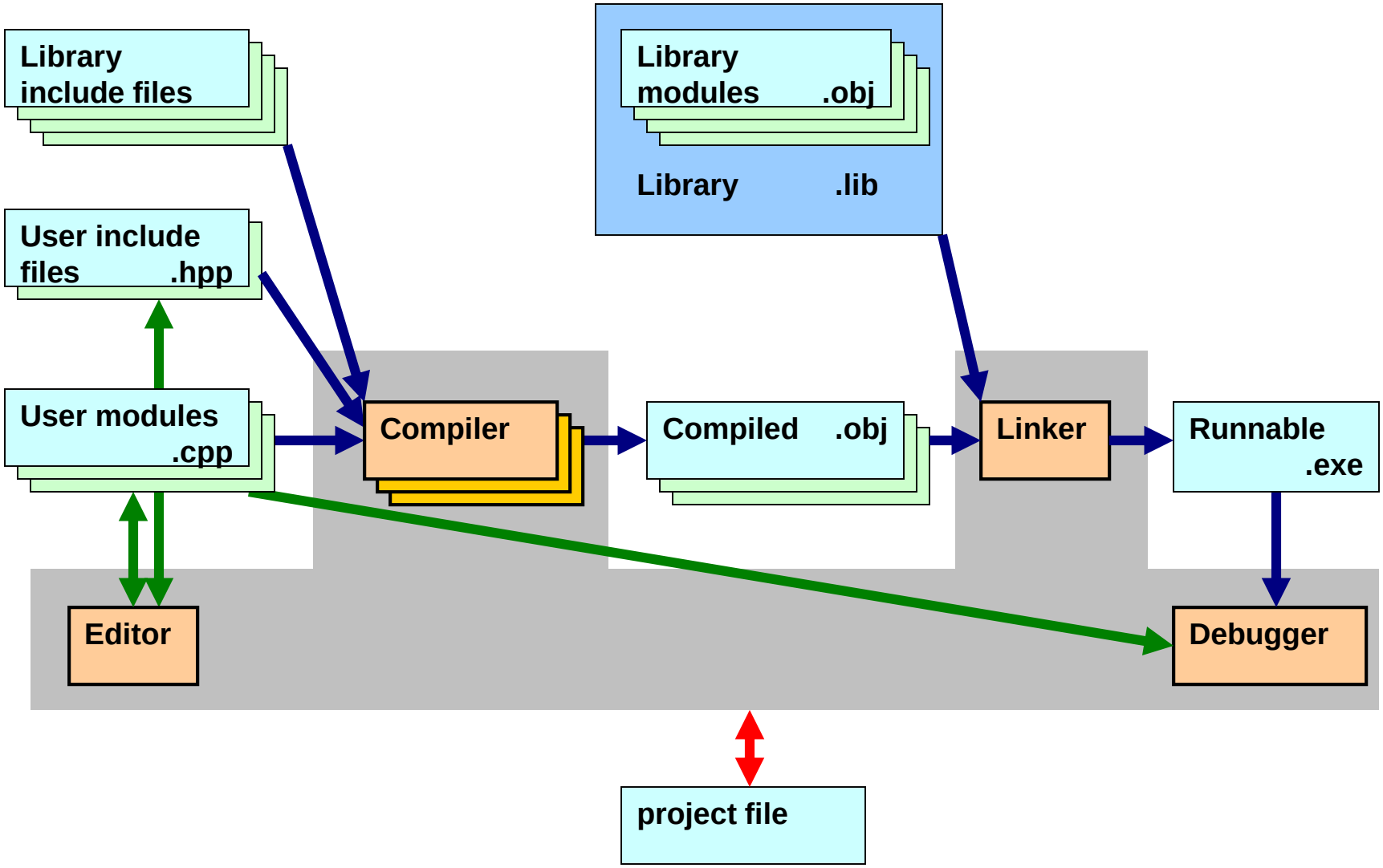


- Modern approach to compiling and linking
  - Function bodies are compiled only to some intermediate code
    - Object modules usually contain no binary code of the target platform (but they still can)
  - Templates are instantiated multiple times
    - Template code must still be located in header files
  - Inline functions are parsed and type-checked multiple times
  - Binary code is generated and optimized only once, during linking
  - In addition, procedure integration may be done across module boundaries

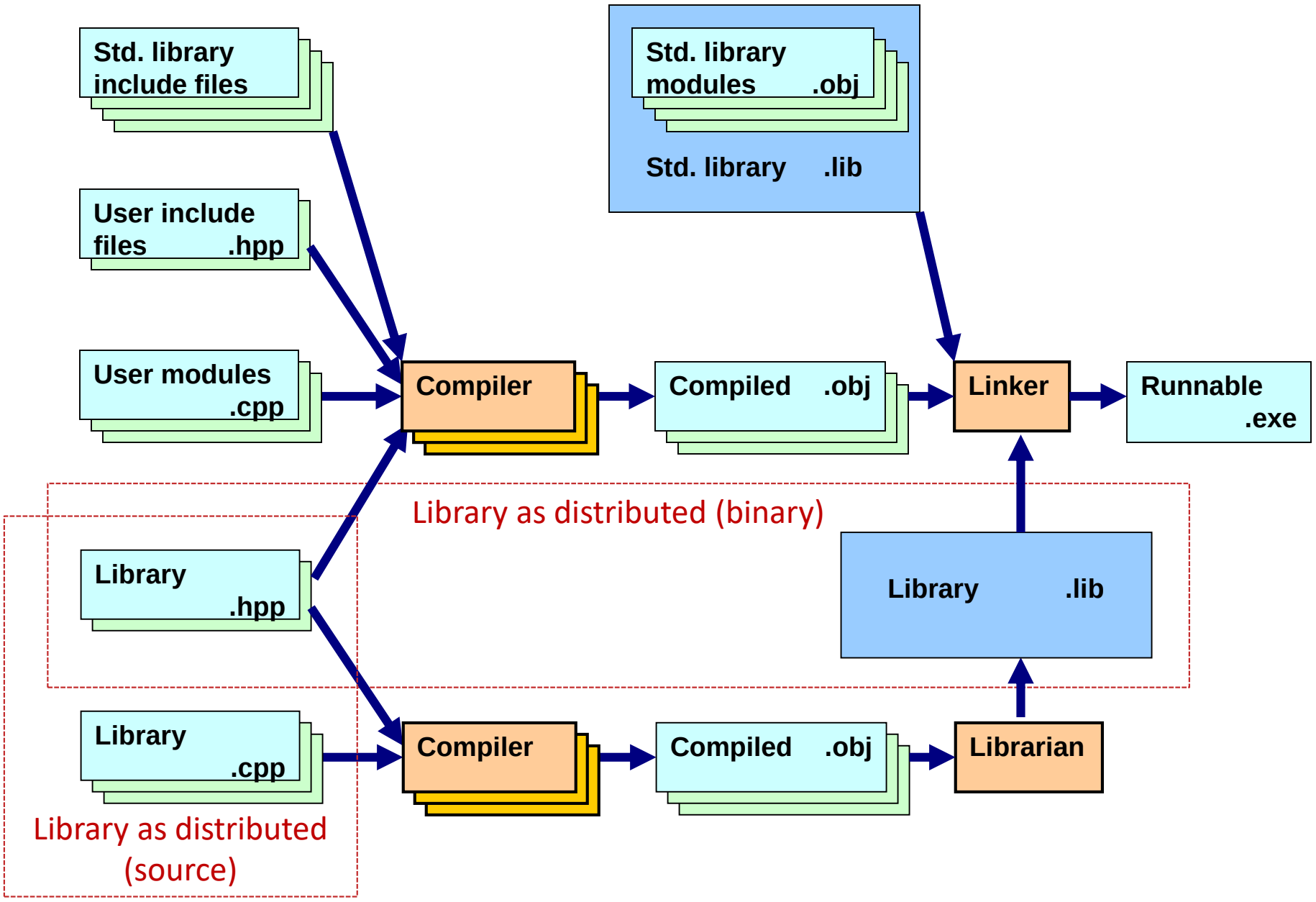
make


Library include files
User include files .hpp
User modules .cpp
Library modules .obj
Library .lib
Compiler
Compiled .obj
Linker
Runnable .exe
Make
makefile

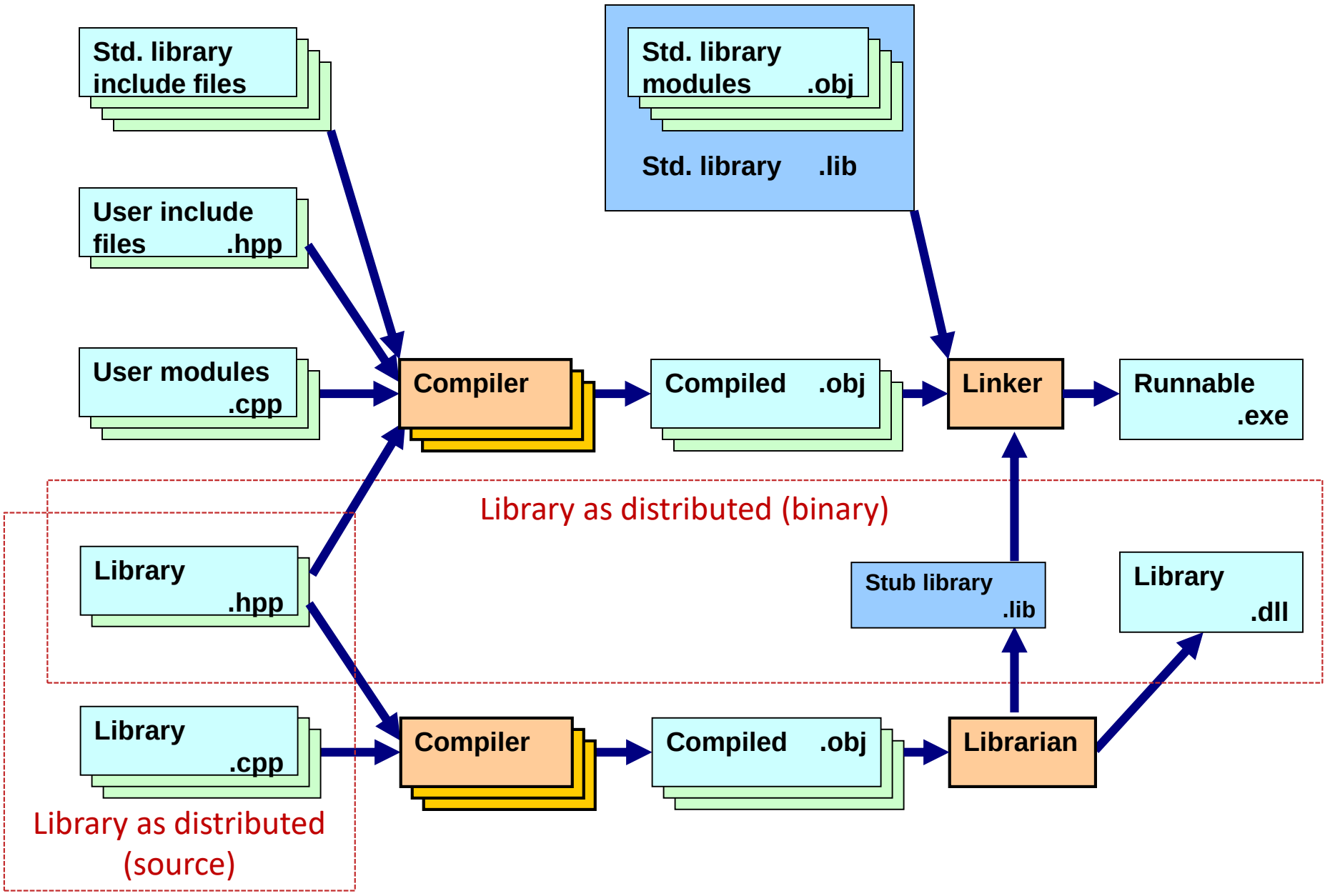
Integrated environment



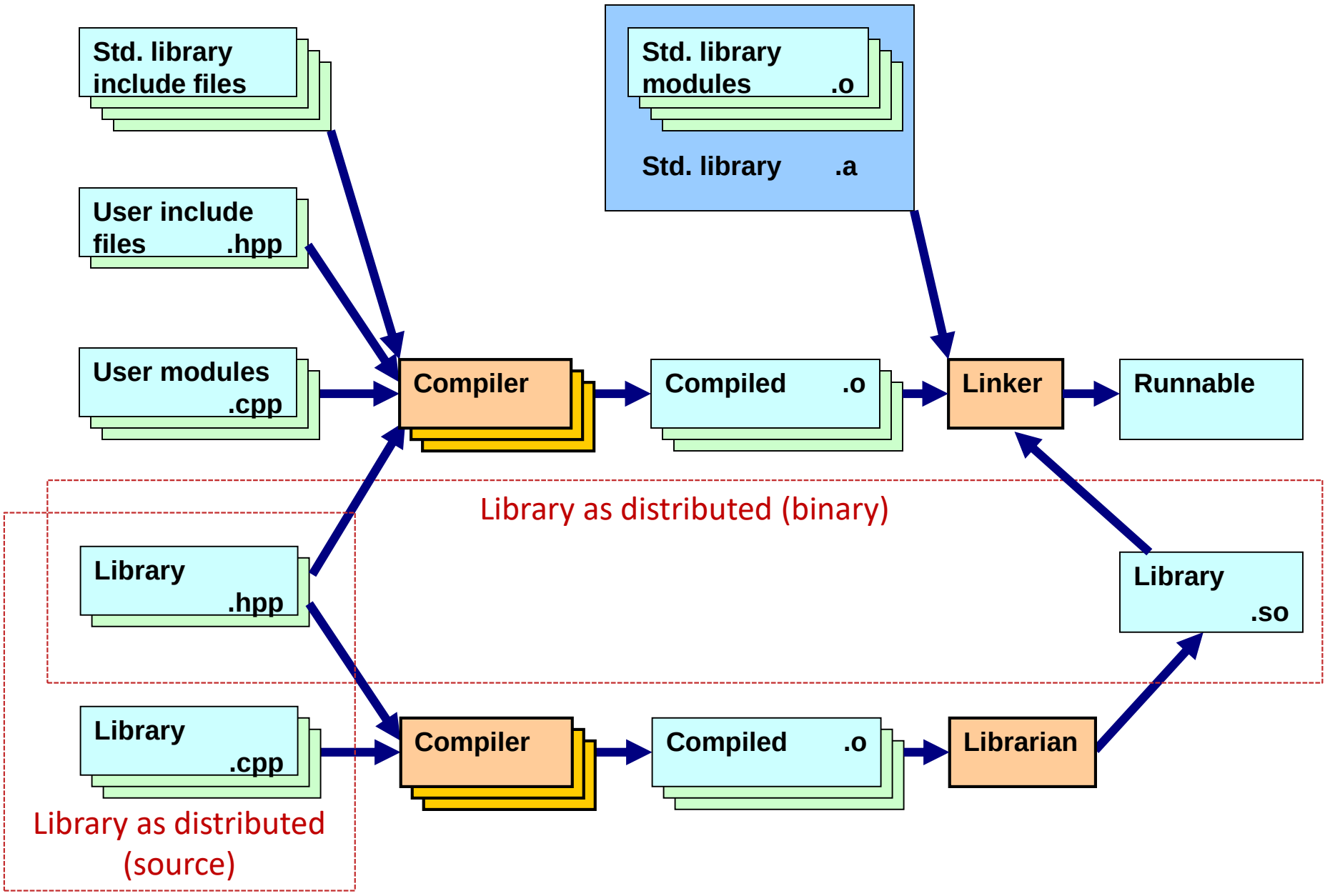
Static libraries



Dynamic libraries (Microsoft)



Dynamic libraries (Linux)



Deklarace a definice

Deklarace a definice

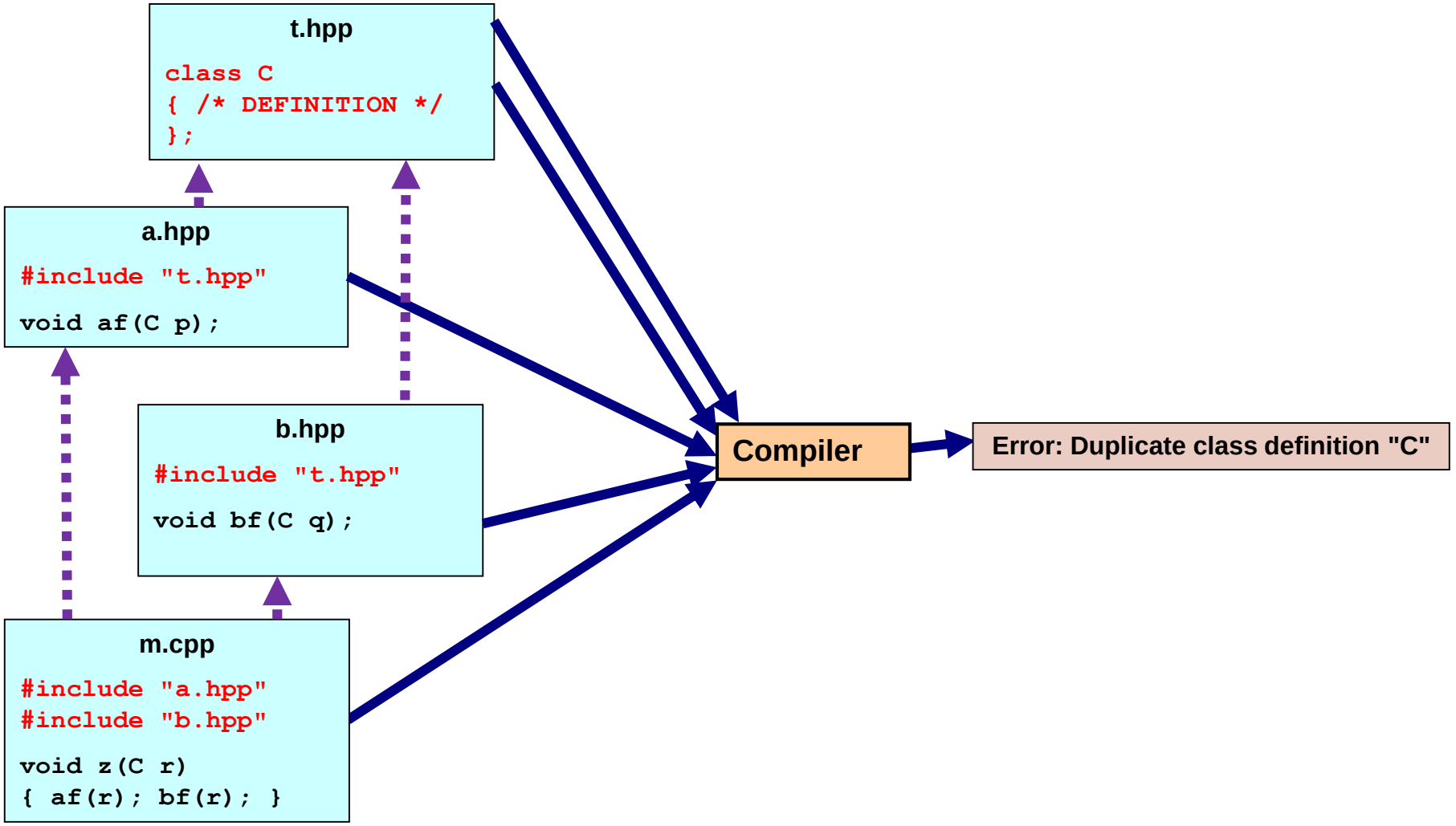
- Deklarace
  - Zápis sdělující, že věc (typ/proměnná/funkce/...) existuje
    - Identifikátor
    - Základní vlastnosti věci
    - Umožňuje překladači přeložit kód, který na věc odkazuje
      - V některých případech je k tomu zapotřebí i definice
- Definice
  - Zápis, který určuje všechny vlastnosti věci
    - Obsah třídy, inicializace proměnné, kód funkce
    - Umožňuje překladači vygenerovat kód a data, která věc reprezentují za běhu
  - Každá definice je i deklarace
- Deklarace umožňují (některá) použití věci bez definice
  - Oddělený překlad modulů
  - Vyřešení cyklických závislostí
  - Zmenšení objemu překládaného zdrojového kódu

Deklarace a definice

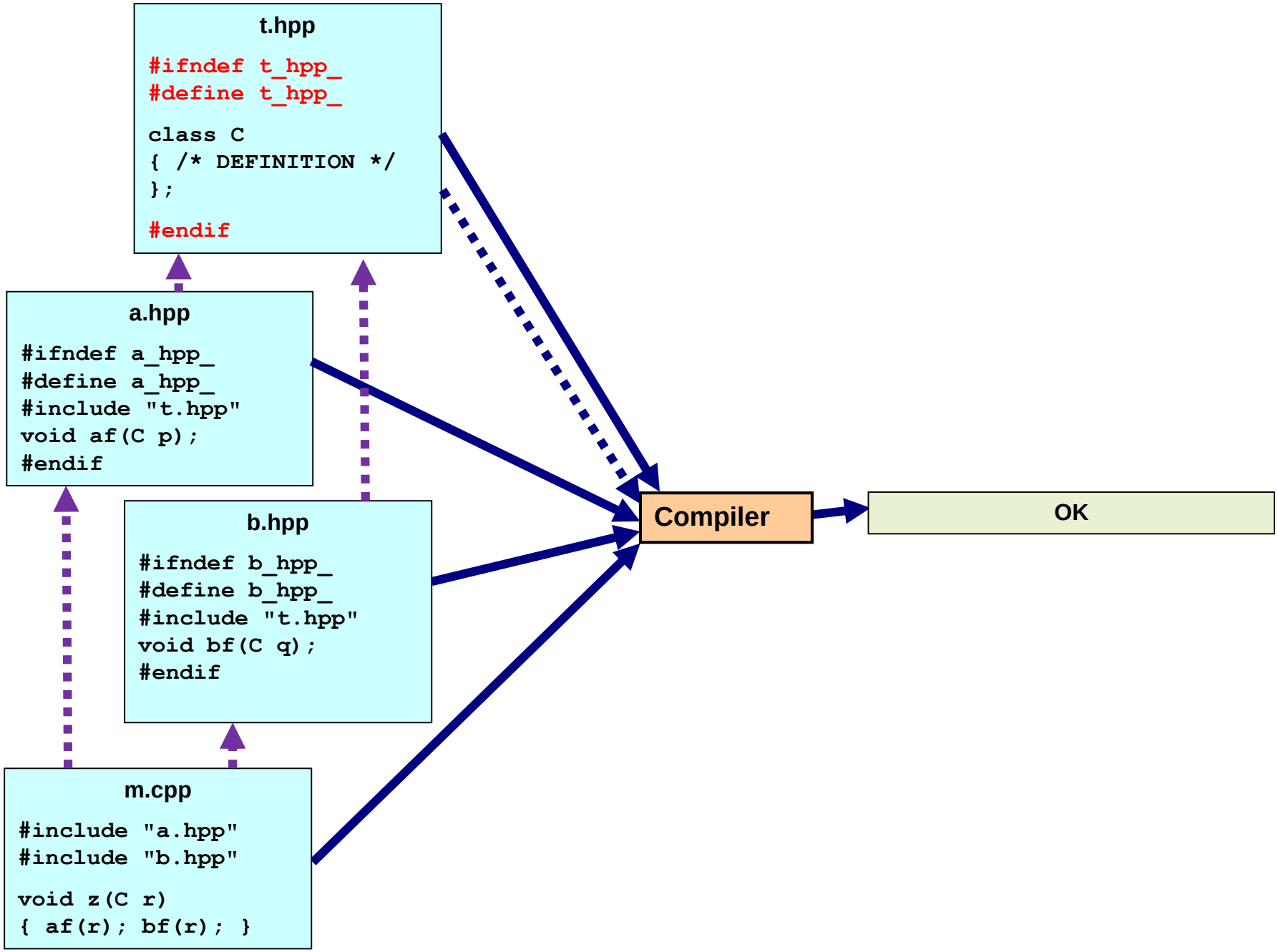
- One-definition rule #1:
  - Jedna *překladová jednotka...*
    - (*modul*, tj. jedno .cpp včetně inkludovaných .hpp)
    - tak, jak ji vidí překladač
  - ... smí obsahovat nejvýše jednu definici věci

- One-definition rule #2:
  - Program...
    - (tj. .exe včetně připojených .dll)
  - ... smí obsahovat nejvýše jednu definici proměnné nebo non-inline funkce
    - Definice třídy, typu či inline funkce se v různých modulech opakovat smějí (typicky vložením téhož .hpp souboru)
      - Nejsou-li opakované definice totožné, nebo nesouhlasí-li definice s deklarací, program je nekorektní
    - Diagnostika na úrovni programu není normou požadována a překladače/linkery ji dělají jen v jednoduchých případech

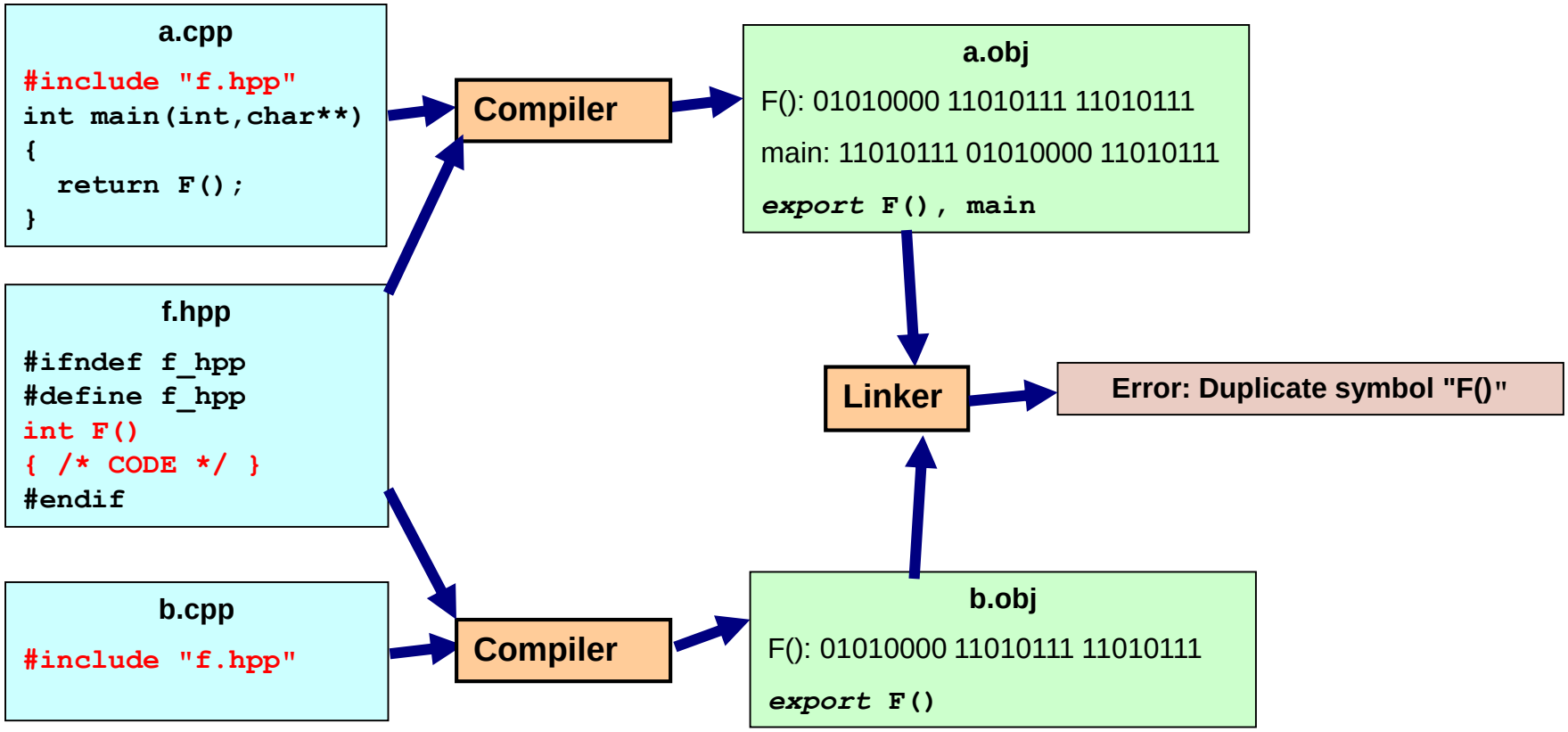
ODR #1 violation



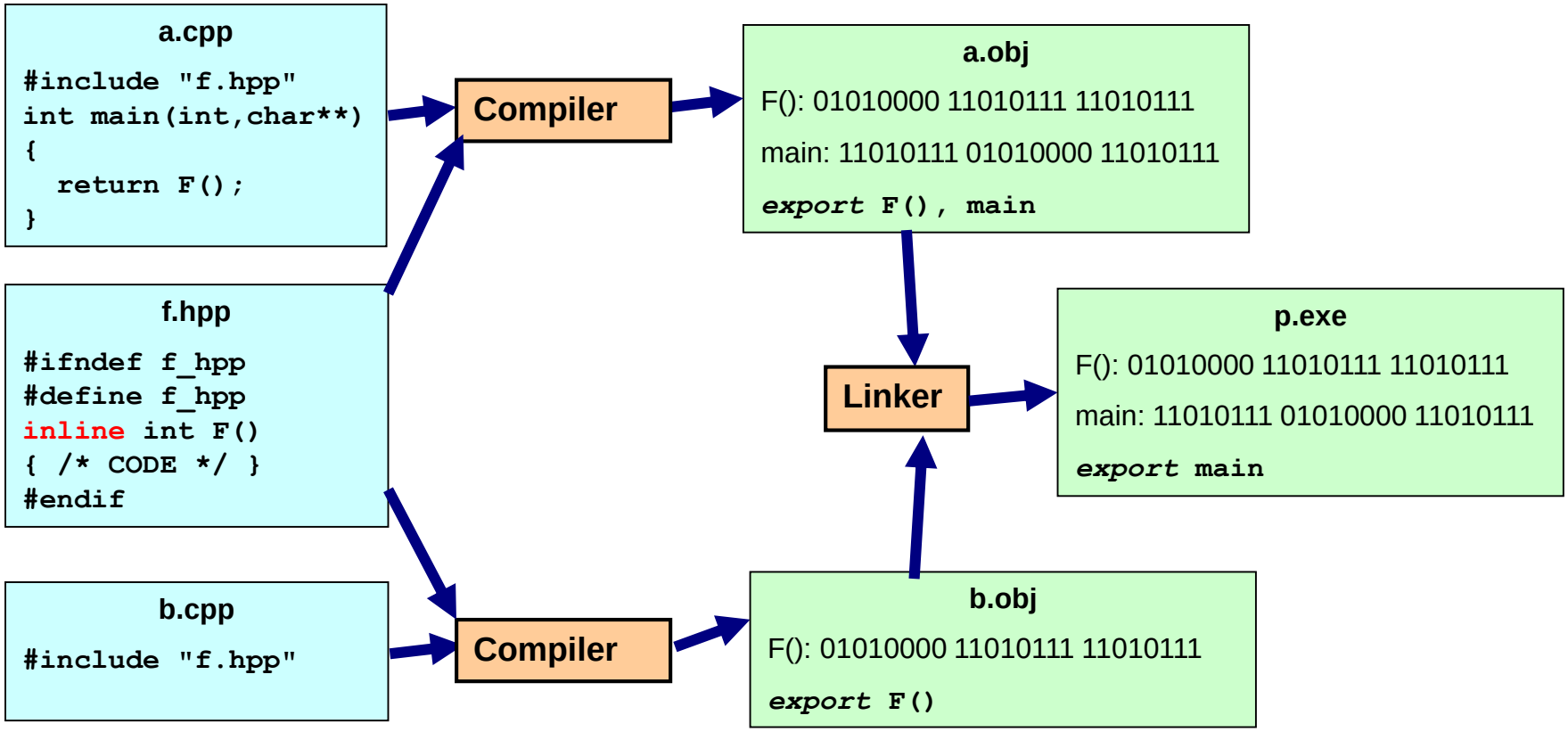
ODR #1 protection



ODR #2 violation



ODR #2 protection



Placement of declarations

- Every name must be declared **before** its first use
  - In every *translation unit* which uses it
  - "Before" refers to the text produced by inclusion and conditional compilation directives
- Special handling of member function bodies
  - Compilation of the body of a member function...
    - if the body is present inside its class definition
  - ... is delayed to the end of its class definition
    - thus, declarations of all class members are visible to the body
- The placement of declaration defines the scope of the name
  - Declaration always uses an unqualified name
- Exception: Friend functions
  - Friend function declaration inside a class may declare the name outside the class (if not already defined)

Placement of declarations

```
class C {
public:
   D f1();          // error: D not declared yet
   int f2() { D x; return x.f3(); }    // OK, compilation delayed
   class D {
   public:
      int f3();
   };
   friend C f4();  // declares global f4 and makes it friend
private:
   int m_;
};
C::D C::f1() { return D{}; }          // qualified name C::D required outside C
int C::D::f3() { return 0; }          // this could be static member function
void C::f5() {}    // error: cannot declare outside the required scope
C f4() { C x; x.m_ = 1; return x; }  // friends may access private members
```

Placement of definitions

- Type alias (typedef/using), enumeration type, constant
  - Must be defined **before** first use (as seen after preprocessing)
- Class/struct
  - Class/struct C must be defined **before** its first **non-trivial** use:
    - (member) variable definition of type C, inheriting from class C
    - creation/copying/moving/destruction of an object of type C
    - access to any member of C
  - **Trivial** use is satisfied with a declaration
    - constructing complex types from C
    - *declaring* functions accepting/returning C
    - manipulating with pointers/references to C
- Inline function
  - must be defined anywhere in **each** translation unit which contains a call
    - the definition is typically placed in a .hpp file
- Non-inline function, global/static variable
  - must be defined exactly **once** in the program (if used)
    - the definition is placed in a .cpp file

Základní pravidla pro .cpp/.hpp

- .hpp – "hlavičkové soubory"
  - Ochrana proti opakovanému include

```
#ifndef myfile_hpp_
#define myfile_hpp_
/* … */
#endif
```

  - Vkládají se direktivou s uvozovkami

```
#include "myfile.hpp"
```

    - Direktiva s úhlovými závorkami je určena pro (standardní) knihovny

```
#include <iostream>
```

  - Direktivy #include používat vždy na začátku souboru (po ifndef+define)
  - Soubor musí být samostatný: vše, co potřebuje, inkluduje sám

- .cpp - "moduly"
  - Zařazení do programu pomocí projektu/makefile
    - Nikdy nevkládat pomocí #include
    - Jeden modul ≈ jeden .cpp

Základní pravidla pro .cpp/.hpp

- .hpp – "hlavičkové soubory"
  - Deklarace/definice typů a tříd
  - Implementace funkcí a metod malého rozsahu
    - Funkce a metody mimo třídy označeny "inline"

```
inline int max( int a, int b) { return a > b ? a : b; }
```

  - Hlavičky globálních funkcí *většího než malého rozsahu*

```
int big_function( int a, int b);
```

  - Externí deklarace globálních proměnných

```
extern int x;
```

    - Raději nepoužívat - lepší je použít singleton
  - Veškerý generický kód (šablony tříd a funkcí)
    - Jinak jej překladače neumí použít

- .cpp - "moduly"
  - Implementace funkcí a metod *většího než malého rozsahu*
    - Včetně "main"
  - Definice globálních a statických proměnných
    - Včetně jejich inicializace

```
int x = 729;
```

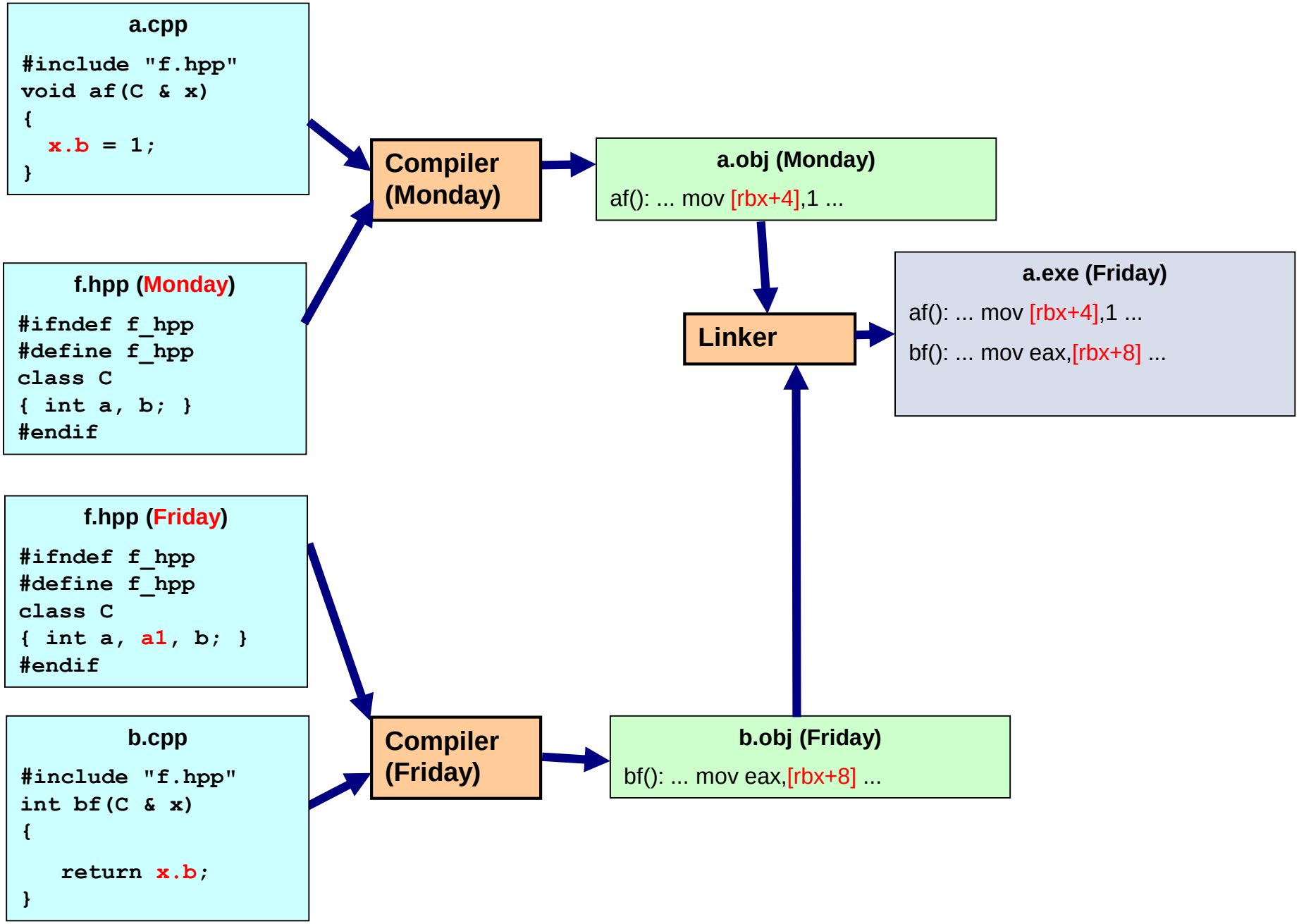
Vzájemné závislosti v kódu

- Všechny identifikátory musejí být deklarovány před prvním použitím
  - Překladač čte zdrojové soubory jedním průchodem
  - Výjimka: Těla metod jsou analyzována až na konci třídy
    - Zevnitř metod lze používat položky deklarované později
  - Pro generický kód platí složitější, ale obdobná pravidla
- Cyklické závislosti je nutné rozbít rozdělením na deklaraci a definici

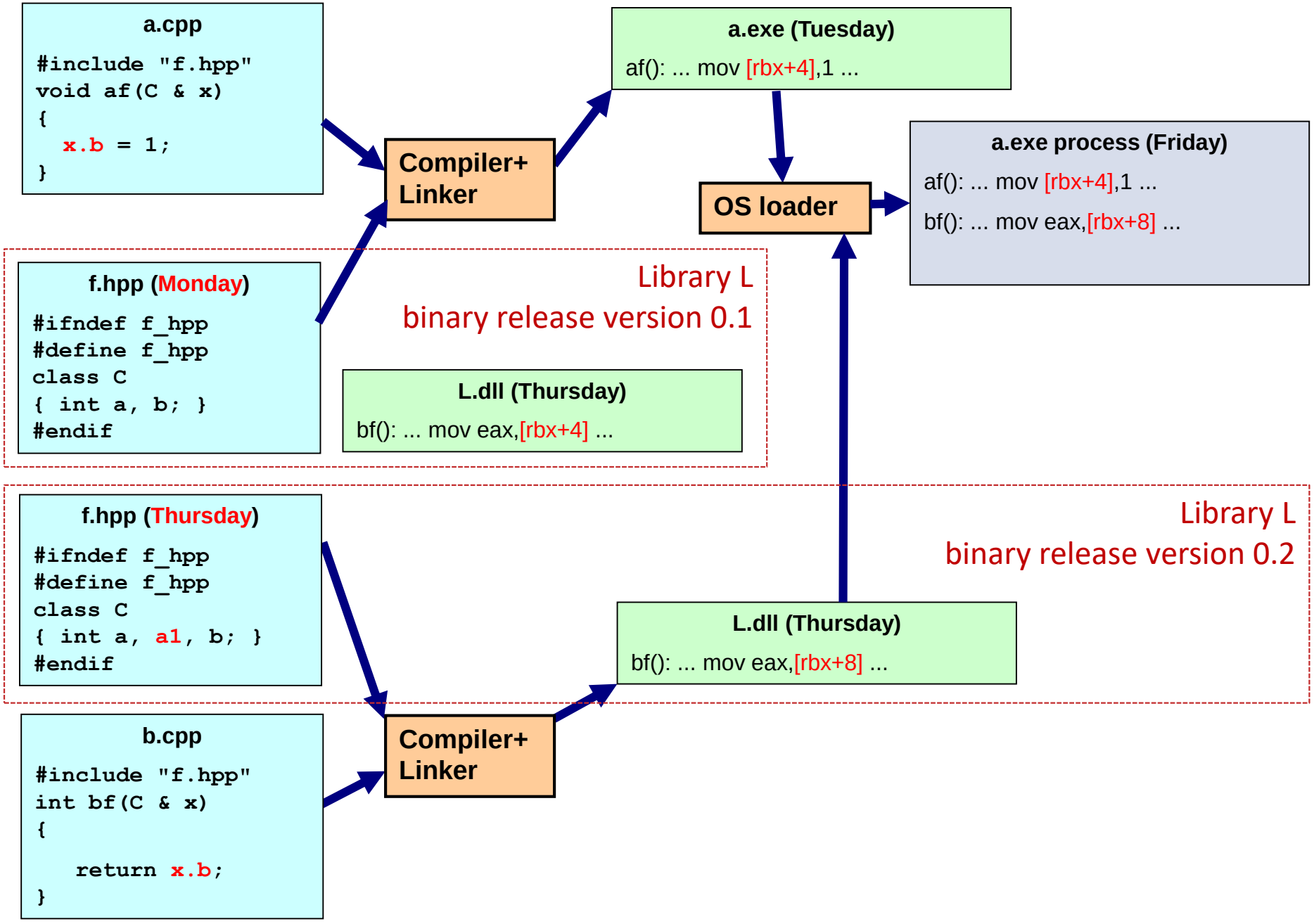
```
class one;

class two {

  std::shared_ptr< one> p_;

};

class one : public two

{};
```

  - Nedefinovaná deklarovaná třída má omezené možnosti použití
    - Nelze použít jako předek, typ položky/proměnné, new, sizeof apod.

Version mismatch



Version mismatch



Deklarace a definice tříd a typů

	Deklarace	Definice
Výčtový typ	`enum E;`	`enum E { E_ZERO, E_ONE };`

Třída	`class A;`	`class A {`
		`  ...`
		`};`

Struktura(téměř ekvivalentní s class)	`struct A;`	`struct A {`
		`  ...`
		`};`

Unie (v C++ téměř nepoužitelné)	`union A;`	`union A {`
		`  ...`
		`};`

Pojmenovaný typ		`typedef A A2;`
		`typedef A * AP;`
		`typedef std::shared_ptr< A> AS;`
		`typedef A AA[ 10];`
		`typedef A (* AFP2)();`
		`typedef std::vector< A> AV;`
		`typedef AV::iterator AVI;`

Pojmenovaný typ (C++11)		`using A2 = A;` `using AFP2 = A (*)();`

Deklarace a definice funkcí

non-inline	Deklarace (.hpp nebo .cpp)	Definice (.cpp)
Global function	`int f( int, int);`	`int f( int p, int q)` `{ return p + q;}`

Static member function	`class A {`
	`  static int f( int p);`
	`};`
		`int A::f( int p)`
		`{ return p + 1;`
		`}`

Nonstatic member function	`class A {`
	`  int f( int p);`
	`};`
		`int A::f( int p)`
		`{ return p + 1;`
		`}`

Virtual member function	`class A {`
	`  virtual int f( int p);`
	`};`
		`int A::f( int)`
		`{ return 0;`
		`}`

inline	**Deklarace (.hpp nebo .cpp)**	**Definice (.hpp nebo .cpp)**

Global inline function		`inline int f( int p, int q)`
		`{ return p + q;`
		`}`

Nonstatic inline member fnc (a)	`class A {`
	`  int f( int p);`
	`};`
		`inline int A::f( int p)`
		`{ return p + 1;`
		`}`

Nonstatic inline member fnc (b)		`class A {`
		`  int f( int p) { return p+1;}`
		`};`

Deklarace a definice proměnných

	Declaration	Definition
Global variable	`extern int x, y, z;`	`int x; int y = 729; int z(729);` `int u{729};` C++11

Static member variable	`class A {`
	`  static int x, y, z;`
	`};`
		`int A::x; int A::y = 729;`
		`int A::z( 729);`
		`int A::z{ 729};` C++11

Constant		`class A {`
		`  static const int x = 729;`
		`};`

Static local variable		`void f() {`
		`  static int x;`
		`  static int y = 7, z( 7);`
		`  static int u{ 7};` C++11
		`}`

Nonstatic member variable		`class A {`
		`  int x, y;`
		`};`

Nonstatic local variable		`void f() {`
		`  int x;`
		`  int y = 7, z( 7);`
		`  int u{ 7};` C++11
		`};`

Často používané datové typy

Vybrané číselné typy

`bool`	false, true
`char`	character (ASCII, 8 bit)
`std::wchar_t`	character (Unicode, 16/32 bit)
`int`	signed integer (~32 bit)
`unsigned`	unsigned integer (~32 bit)
`std::uint_least64_t`	unsigned integer (nejmenší >= 64 bit)
`std::int_fast8_t`	signed integer (nejrychlejší >= 8 bit)
`long long`	extra large signed integer (~64 bit)
`unsigned long long`	extra large unsigned integer (~64 bit)
`std::size_t`	unsigned integer dostatečný pro velikost pole (32/64 bit)
`std::ptrdiff_t`	signed integer dostatečný pro indexování pole (32/64 bit)
`double`	"double precision" floating-point number (Intel: 64 bit)
`long double`	extended precision floating-point number (Intel: 80 bit)
`std::complex<double>`	complex number of double precision

Důležité nečíselné typy

`std::string`	string (obsahující char)
`std::wstring`	string (obsahující std::wchar_t)
`std::istream`	input stream (nad char)
`std::wistream`	input stream (nad std::wchar_t)
`std::ostream`	output stream (nad char)
`std::wostream`	output stream (nad std::wchar_t)
`struct T { … }`	struktura (téměř ekvivalent class)
`std::pair<T1,T2>`	dvojice T1, T2
`std::tuple<T1,...,Tn>`	n-tice T1,...,Tn
`std::array<T,n>`	pole pevné velikosti prvků typu T
`std::vector<T>`	pole proměnlivé velikosti prvků typu T
`std::list<T>`	obousměrný spojový seznam prvků typu T
`std::map<K,T>`	vyhledávací strom s klíčem K a prvky typu T
`std::multimap<K,T>`	vyhledávací strom s opakováním
`std::unordered_map<K,T>`	hašovací tabulka s klíčem K a prvky typu T
`std::unordered_multimap<K,T>`	hašovací tabulka s opakováním

Exception handling

Exception handling

- Mechanismus výjimek
  - Start: příkaz throw
  - Cíl: try-catch blok
    - Určen za běhu
  - Skok může opustit proceduru
    - Proměnné korektně zaniknou voláním destruktorů
  - Předává hodnotu libovolného typu
    - Typ hodnoty se podílí na určení cíle skoku
    - Obvykle se používají pro tento účel zhotovené třídy
    - Mechanismus výjimek respektuje hierarchii dědičnosti

```
class AnyException { /*...*/ };
class WrongException
 : public AnyException { /*...*/ };
class BadException
 : public AnyException { /*...*/ };
void f()
{
 if ( something == wrong )
  throw WrongException( something);
 if ( anything != good )
  throw BadException( anything);
}
void g()
{
 try {
  f();
 }
 catch ( const AnyException & e1 ) {
  /*...*/
 }
}
```

Exception handling

- Mechanismus výjimek
  - Start: příkaz throw
  - Cíl: try-catch blok
    - Určen za běhu
  - Skok může opustit proceduru
    - Proměnné korektně zaniknou voláním destruktorů
  - Předává hodnotu libovolného typu
    - Typ hodnoty se podílí na určení cíle skoku
    - Obvykle se používají pro tento účel zhotovené třídy
    - Mechanismus výjimek respektuje hierarchii dědičnosti
    - Hodnotu není třeba využívat

```
class AnyException { /*...*/ };
class WrongException
  : public AnyException { /*...*/ };
class BadException
  : public AnyException { /*...*/ };
void f()
{
  if ( something == wrong )
    throw WrongException();
  if ( anything != good )
    throw BadException();
}
void g()
{
  try {
    f();
  }
  catch ( const AnyException &) {
    /*...*/
  }
}
```

Exception handling

- Mechanismus výjimek
  - Start: příkaz throw
  - Cíl: try-catch blok
    - Určen za běhu
  - Skok může opustit proceduru
    - Proměnné korektně zaniknou voláním destruktorů
  - Předává hodnotu libovolného typu
    - Typ hodnoty se podílí na určení cíle skoku
    - Obvykle se používají pro tento účel zhotovené třídy
    - Mechanismus výjimek respektuje hierarchii dědičnosti
    - Hodnotu není třeba využívat
    - Existuje univerzální catch blok

```
class AnyException { /*...*/ };
class WrongException
  : public AnyException { /*...*/ };
class BadException
  : public AnyException { /*...*/ };
void f()
{
  if ( something == wrong )
    throw WrongException();
  if ( anything != good )
    throw BadException();
}
void g()
{
  try {
    f();
  }
  catch (...) {
    /*...*/
  }
}
```

Exception handling

- Fáze zpracování výjimky
  - Vyhodnocení výrazu v příkaze throw
    - Hodnota je uložena "stranou"
  - Stack-unwinding
    - Postupně se opouštějí bloky a funkce, ve kterých bylo provádění vnořeno
    - Na zanikající lokální a pomocné proměnné jsou volány destruktory
    - Stack-unwinding končí dosažením try-bloku, za kterým je catch-blok odpovídající typu výrazu v příkaze throw
  - Provedení kódu v catch-bloku
    - Původní hodnota throw je stále uložena pro případné pokračování:
    - Příkaz throw bez výrazu pokračuje ve zpracování téže výjimky počínaje dalším catch-blokem - začíná znovu stack-unwinding
  - Zpracování definitivně končí opuštěním catch-bloku
    - Běžným způsobem nebo příkazy return, break, continue, goto
    - Nebo vyvoláním jiné výjimky

Exception handling

- Zhmotněné výjimky
  - std::exception_ptr je chytrý ukazatel na objekt výjimky
    - Objekt zanikne při zániku posledního ukazatele
  - std::current_exception()
    - Vrací aktuálně řešenou výjimku
  - std::rethrow_exception(p)
    - Vyvolává uloženou výjimku
  - Tento mechanismus umožňuje odložit ošetřování výjimky, zejména:
    - Propagace výjimky do jiného vlákna
    - Řešení výjimek v promise/future

C++11

```
std::exception_ptr p;

void g()
{
 try {
  f();
 }
 catch (...) {
  p = std::current_exception();
 }
}

void h()
{
 std::rethrow_exception( p);
}
```

Exception handling

Použití mechanismu výjimek

- Vyvolání a zpracování výjimky je relativně časově náročné
  - Používat pouze pro chybové nebo řídké stavy
  - Např. nedostatek paměti, ztráta spojení, chybný vstup, konec souboru
- Připravenost na výjimky také něco (málo) stojí
  - Za normálního běhu je třeba zařídit, aby výjimka dokázala najít cíl a zrušit proměnné
  - Výjimky se týkají i procedur, ve kterých není ani throw, ani try-blok
  - Většina kompilátorů umí překládat ve dvou režimech "s" a "bez"
  - Celý spojovaný program musí být přeložen stejně

Exception handling

- Standardní výjimky
  - <stdexcept>
  - Všechny standardní výjimky jsou potomky třídy exception
    - metoda what() vrací řetězec s chybovým hlášením
  - **bad_alloc**: vyvolává operátor new při nedostatku paměti
    - V režimu "bez výjimek" new vrací nulový ukazatel
  - **bad_cast**, **bad_typeid**: Chybné použití RTTI
  - Odvozené z třídy logic_error:
    - **domain_error, invalid_argument, length_error, out_of_range**
    - vyvolávány např. funkcí vector::operator[]
  - Odvozené z třídy runtime_error:
    - **range_error, overflow_error, underflow_error**

Exception handling

- Standardní výjimky
  - <stdexcept>
  - Všechny standardní výjimky jsou potomky třídy exception
    - metoda what() vrací řetězec s chybovým hlášením
  - bad_alloc: vyvolává operátor new při nedostatku paměti
    - V režimu "bez výjimek" new vrací nulový ukazatel
  - bad_cast, bad_typeid: Chybné použití RTTI
  - Odvozené z třídy logic_error:
    - domain_error, invalid_argument, length_error, out_of_range
    - vyvolávány např. funkcí vector::operator[]
  - Odvozené z třídy runtime_error:
    - range_error, overflow_error, underflow_error
  - Aritmetické ani ukazatelové operátory na vestavěných typech NEHLÁSÍ běhové chyby prostřednictvím výjimek
    - např. dělení nulou nebo dereference nulového ukazatele

Exception specifications

- Exception specifications
  - U každé funkce (operátoru, metody) je možno určit seznam výjimek, kterými smí být ukončena
    - Na výjimky ošetřené uvnitř funkce se specifikace nevztahuje
    - Pokud není specifikace uvedena, povoleny jsou všechny výjimky
    - Specifikace respektuje dědičnost, to jest automaticky povoluje i všechny potomky uvedené třídy

```
void a()
{
  /* tahle smí všechno */
}

void b() throw ()
{
  /* tahle nesmí nic */
}

void c() throw ( std::bad_alloc)
{
  /* tahle smí std::bad_alloc */
}

void d() throw ( std::exception, MyExc)
{
  /* tahle smí potomky
    std::exception a MyExc */
}
```

Exception specifications

- Exception specifications
  - Kompilátor zajistí, že nepovolená výjimka neopustí funkci:
    - Pokud by se tak mělo stát, volá se unexpected()
    - unexpected() smí vyvolat "náhradní" výjimku
    - Pokud ani náhradní výjimka není povolena, zkusí se vyvolat std::bad_exception
    - Pokud ani std::bad_exception není povoleno, volá se terminate() a program končí

Exception specifications

- Exception specifications
  - Kompilátor zajistí, že nepovolená výjimka neopustí funkci
  - Toto je běhová kontrola
    - Kompilátor smí vydávat nejvýše varování
    - Funkce smí volat jinou, která by mohla vyvolat nepovolenou výjimku (ale nemusí)

```
void f() throw ( std::exception)
{
}

void g() throw ()
{
  f(); /* tohle se smí */
}
```

Exception specifications

- Exception specifications

  - throw(T) specifikace se příliš nepoužívaly

  - C++11 definuje novou syntaxi
    - noexcept
    - noexcept(c) kde c je Booleovský konstantní výraz

C++11

```
void f() noexcept
{
}

template< typename T>
void g( T & y)
 noexcept(
 std::is_nothrow_copy_constructible
  < T>::value)
{
 T x = y;
}
```

Exception-safe programming

Exception-safe programming

- Používat throw a catch je jednoduché
- Těžší je programovat běžný kód tak, aby se choval korektně i za přítomnosti výjimek
  - Exception-safety
  - Exception-safe programming

```
void f()
{
  int * a = new int[ 100];
  int * b = new int[ 200];
  g( a, b);
  delete[] b;
  delete[] a;
}
```

- Pokud new int[200] způsobí výjimku, procedura zanechá naalokovaný nedostupný blok
- Pokud výjimku vyvolá procedura g, zůstanou dva nedostupné bloky

Exception-safe programming

- Použití chytrých ukazatelů řeší i některé problémy s výjimkami

```
void f()
{
  auto a = std::make_unique< int[]>( 100);
  auto b = std::make_unique< int[]>( 200);
  g( a, b);
}
```

- RAII: Resource Acquisition Is Initialization
  - Konstruktor zabírá prostředky
  - Destruktor je uvolňuje
    - I v případě výjimky

```
std::mutex my_mutex;

void f()
{
  std::lock_guard< std::mutex>
    lock( my_mutex);
  // do something critical here
}
```

Exception-safe programming

- Používat throw a catch je jednoduché
- Těžší je programovat běžný kód tak, aby se choval korektně i za přítomnosti výjimek
  - Exception-safety
  - Exception-safe programming

```
T & operator=( const T & b)
{
 if ( this != & b )
 {
  delete body_;
  body_ = new TBody( b.length());
  copy( body_, b.body_);
 }
 return * this;
}
```

- Pokud new TBody způsobí výjimku, operátor= zanechá v položce body_ původní ukazatel, který již míří na dealokovaný blok
- Pokud výjimku vyvolá procedura copy, operátor zanechá třídu v neúplném stavu

Exception-safe programming

- Pravidla vynucená jazykem

  - Destruktor nesmí skončit vyvoláním výjimky
    - Výjimka může být vyvolána uvnitř, ale musí být zachycena nejpozději uvnitř destruktoru

  - Zdůvodnění:
    - V rámci ošetření výjimek (ve fázi stack-unwinding) se volají destruktory lokálních proměnných
    - Výjimku zde vyvolanou nelze z technických i logických důvodů ošetřit (ztratila by se původní výjimka)
    - Nastane-li taková výjimka, volá se funkce terminate() a program končí

Exception-safe programming

Pravidla vynucená jazykem

- Destruktor nesmí skončit vyvoláním výjimky
  - Výjimka může být vyvolána uvnitř, ale musí být zachycena nejpozději uvnitř destruktoru
- Toto pravidlo jazyka sice platí pouze pro destruktory lokálních proměnných
  - A z jiných důvodů též pro globální proměnné
- Je však vhodné je dodržovat vždy
  - Bezpečnostní zdůvodnění: Destruktory lokálních proměnných často volají jiné destruktory
  - Logické zdůvodnění: Nesmrtelné objekty nechceme

Exception-safe programming

- Pravidla vynucená jazykem

  - Destruktor nesmí skončit vyvoláním výjimky

  - Konstruktor globálního objektu nesmí skončit vyvoláním výjimky
    - Zdůvodnění: Není místo, kde ji zachytit
    - Stane-li se to, volá se terminate() a program končí
    - Jiné konstruktory ale výjimky volat mohou (a bývá to vhodné)

Exception-safe programming

- Pravidla vynucená jazykem

  - Destruktor nesmí skončit vyvoláním výjimky
  - Konstruktor globálního objektu nesmí skončit vyvoláním výjimky
  - Copy-constructor typu v hlavičce catch-bloku nesmí skončit vyvoláním výjimky
    - Zdůvodnění: Catch blok by nebylo možné vyvolat
    - Stane-li se to, volá se terminate() a program končí
    - Jiné copy-constructory ale výjimky volat mohou (a bývá to vhodné)

Exception-safe programming

- Pravidla vynucená jazykem
  - Destruktor nesmí skončit vyvoláním výjimky
  - Konstruktor globálního objektu nesmí skončit vyvoláním výjimky
  - Copy-constructor typu v hlavičce catch-bloku nesmí skončit vyvoláním výjimky

Exception-safe programming

- Poznámka: Výjimky při zpracování výjimky

  - Výjimka při výpočtu výrazu v throw příkaze
    - Tento throw příkaz nebude vyvolán
  - Výjimka v destruktoru při stack-unwinding
    - Povolena, pokud neopustí destruktor
    - Po zachycení a normálním ukončení destruktoru se pokračuje v původní výjimce
  - Výjimka uvnitř catch-bloku
    - Pokud je zachycena uvnitř, ošetření původní výjimky může dále pokračovat (přikazem throw bez výrazu)
    - Pokud není zachycena, namísto původní výjimky se pokračuje ošetřováním nové

Exception-safe programming

- Kompilátory samy ošetřují některé výjimky
  - Dynamická alokace polí
    - Dojde-li k výjimce v konstruktoru některého prvku, úspěšně zkonstruované prvky budou destruovány

Ve zpracování výjimky se poté pokračuje

Exception-safe programming

- Kompilátory samy ošetřují některé výjimky
  - Dynamická alokace polí
    - Dojde-li k výjimce v konstruktoru některého prvku, úspěšně zkonstruované prvky budou destruovány
    - Ve zpracování výjimky se poté pokračuje
  - Výjimka v konstruktoru součásti (prvku nebo předka) třídy
    - Sousední, již zkonstruované součásti, budou destruovány
    - Ve zpracování výjimky se poté pokračuje
    - Uvnitř konstruktoru je možno výjimku zachytit speciálním try-blokem:

```
X::X( /* formální parametry */)
try : Y( /* parametry pro konstruktor součásti Y */)
{ /* vlastní tělo konstruktoru */
} catch ( /* parametr catch-bloku */ ) {
  /* ošetření výjimky v konstruktoru Y i ve vlastním těle */
}
```

    - Konstrukci objektu nelze dokončit
    - Opuštění speciálního catch bloku znamená throw;

Exception-safe programming

- Definice

- (Weak) exception safety
  - Funkce (operátor, konstruktor) je *(slabě) bezpečná*, pokud i v případě výjimky zanechá veškerá data v konzistentním stavu
  - Konzistentní stav znamená zejména:
  - Nedostupná data byla korektně destruována a odalokována
  - Ukazatele nemíří na odalokovaná data
  - Platí další invarianty dané logikou aplikace

Exception-safe programming

- Definice

- (Weak) exception safety
  - Funkce (operátor, konstruktor) je *(slabě) bezpečná*, pokud i v případě výjimky zanechá veškerá data v konzistentním stavu
  - Konzistentní stav znamená zejména:
  - Nedostupná data byla korektně destruována a odalokována
  - Ukazatele nemíří na odalokovaná data
  - Platí další invarianty dané logikou aplikace

- Strong exception safety
  - Funkce je *silně bezpečná*, pokud v případě, že skončí vyvoláním výjimky, zanechá data ve stejném (*pozorovatelném*) stavu, ve kterém byla při jejím vyvolání
  - *Observable state* - chování veřejných metod
  - Nazýváno též "Commit-or-rollback semantics"

Exception-safe programming

- Poznámky

- (Weak) exception safety
  - Tohoto stupně bezpečnosti lze většinou dosáhnout
  - Stačí vhodně definovat nějaký konzistentní stav, kterého lze vždy dosáhnout, a ošetřit pomocí něj všechny výjimky
  - Konzistentním stavem může být třeba nulovost všech položek
  - Je nutné upravit všechny funkce tak, aby je tento konzistentní stav nepřekvapil (mohou na něj ale reagovat výjimkou)

- Strong exception safety
  - Silné bezpečnosti nemusí jít vůbec dosáhnout, pokud je rozhraní funkce navrženo špatně
  - Obvykle jsou problémy s funkcemi s dvojím efektem
  - Příklad: funkce pop vracející odebranou hodnotu

Softwarové inženýrství a C++

Co je to programování

- Algorithms + Data Structures = Programs
  - Niklaus Wirth, 1976

Co je to programování

- Algorithm[illegible]ures = Programs
  - Niklaus W[illegible]

Co je to programování

- Čím proslul Henry Ford?
  - Vynalezl automobil?

Co je to programování

- Čím proslul Henry Ford?
  - Vynalezl automobil?
    - Ne. Karl Benz 1885.
  - Byl prvním výrobcem automobilů?

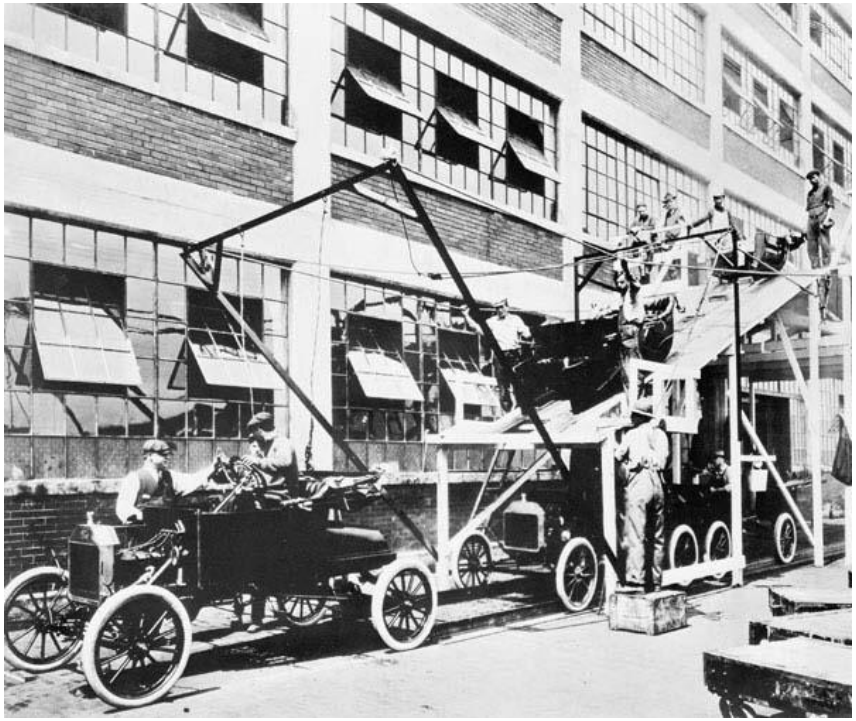
Co je to programování

- Čím proslul Henry Ford?
  - Vynalezl automobil?
    - Ne. Karl Benz 1885.
  - Byl prvním výrobcem automobilů?
    - Ne. Panhard et Levassor 1887.
  - Dokázal při výrobě automobilů využít pracovní sílu lidí, kteří by sami automobil postavit nedokázali.

- Úkolem dobrého programátora je vytvořit kód, který dokážou používat i horší programátoři
  - V C++ je to dvojnásobně důležité

Automobil

- První automobil
  - Karl Benz 1885
- Pásová výroba automobilů
  - Henry Ford 1913



Software

- První programovatelný počítač
  - EDSAC 1949
- Systematická výroba software
  - ???

Co je to programování

Automobil

- První automobil
  - Karl Benz 1885
- Pásová výroba automobilů
  - Henry Ford 1913
  - EU27(2010, včetně výrobců částí):
    2 172 000 zaměstnanců
    95 269 000 000 EUR mzdy

Software

- První programovatelný počítač
  - EDSAC 1949
- Systematická výroba software
  - ???
  - EU27(2010, včetně IT služeb):
    481 000 zaměstnanců
    15 783 000 000 EUR mzdy

Co je to programování

Automobil

- První automobil
  - Karl Benz 1885
- Pásová výroba automobilů
  - Henry Ford 1913
- Kdyby byly běžné automobily stejně spolehlivé jako běžný software, byli bychom dnes všichni mrtví

Software

- První programovatelný počítač
  - EDSAC 1949
- Systematická výroba software
  - ???

Co je to programování

Automobil

- První automobil
  - Karl Benz 1885
- Pásová výroba automobilů
  - Henry Ford 1913
- Kdyby byly běžné automobily stejně spolehlivé jako běžný software, byli bychom dnes všichni mrtví

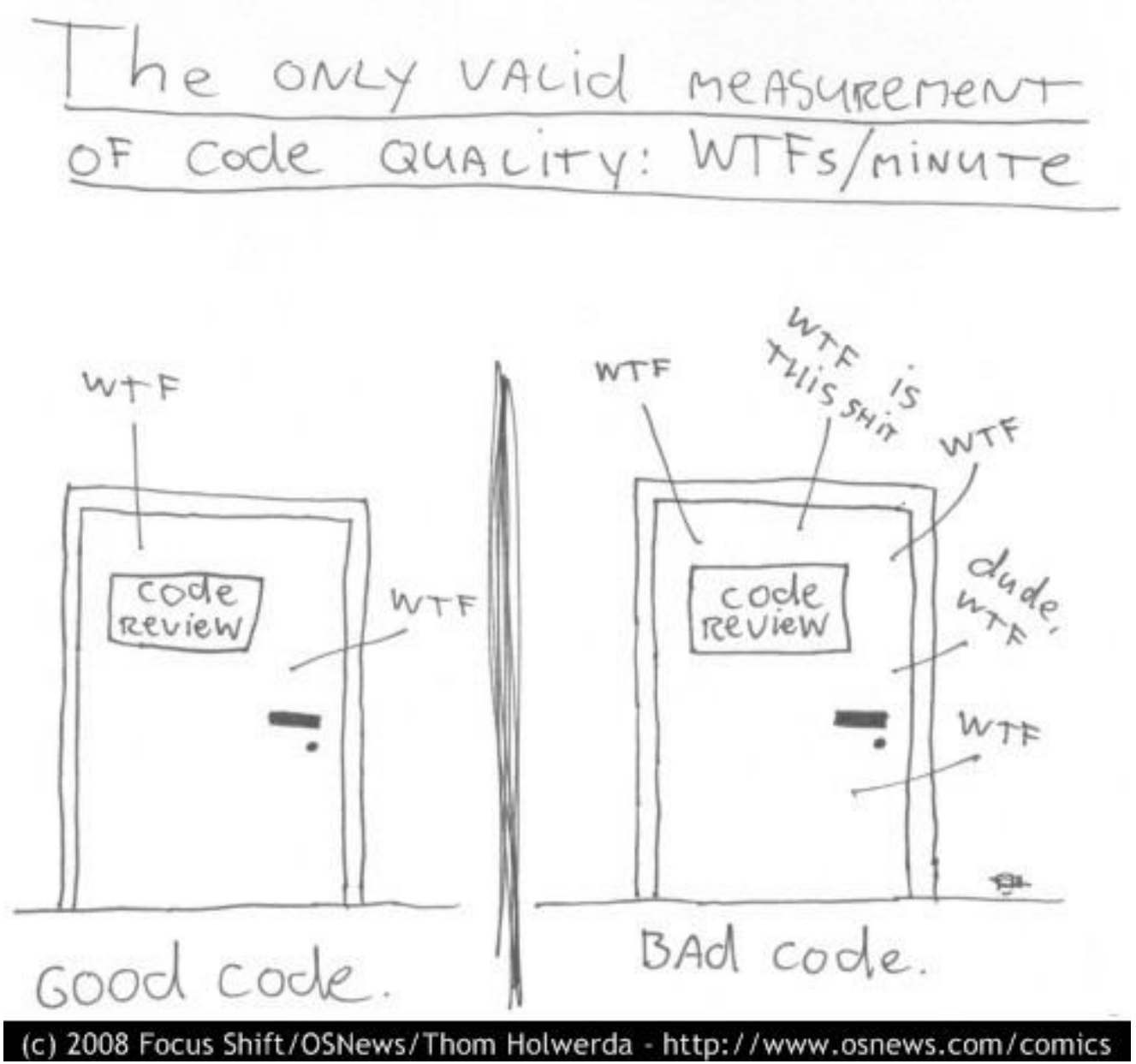
Software

- První programovatelný počítač
  - EDSAC 1949
- Systematická výroba software
  - ???
- 2010: Každý automobil obsahuje nejméně 30 vestavěných počítačů
  - Většina programována v C/C++
  - Spolehlivý software existuje!

Co je to programování

- Každý program se dříve či později stane kolektivním dílem...
  - ...nebo zmizí jako neúspěšný
- Každý programátor by měl počítat s tím, že jeho dílo bude používat někdo cizí
  - Žádná překvapení, žádné exhibice geniality
  - Dodržování konvencí, analogie dobře známých rozhraní

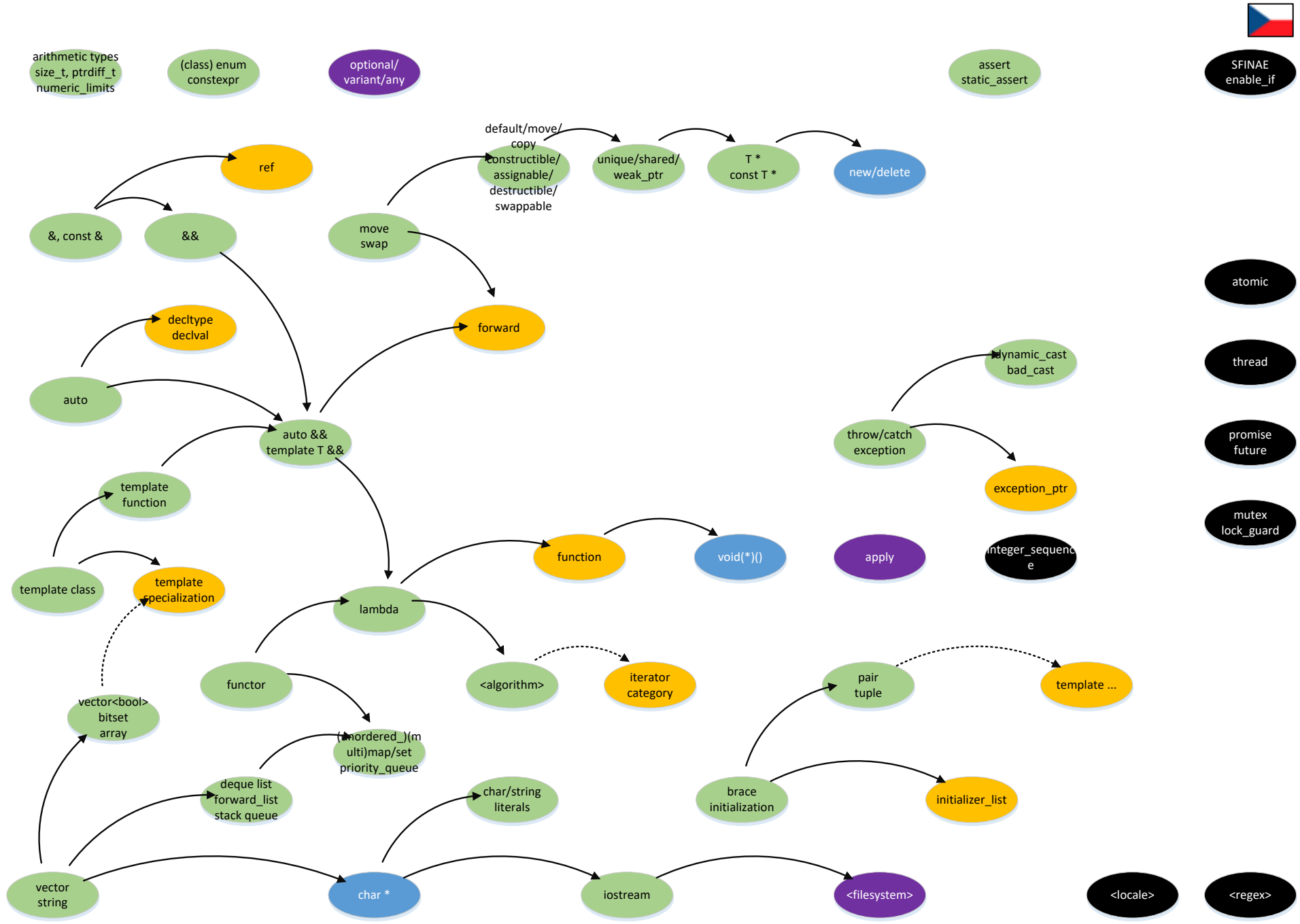
Žádná překvapení



Co je to programování

- Algorithms + Data Structures + Best Practices = Programs

C++


arithmetic types
size_t, ptrdiff_t
numeric_limits
(class) enum
constexpr
optional/
variant/any
assert
static_assert
SFINAE
enable_if
ref
default/move/
copy
constructible/
assignable/
destructible/
swappable
unique/shared/
weak_ptr
T *
const T *
new/delete
&, const &
&&
move
swap
atomic
decltype
declval
forward
thread
auto
auto &&
template T &&
dynamic_cast
bad_cast
throw/catch
exception
promise
future
exception_ptr
template
function
mutex
lock_guard
function
void(*)()
apply
integer_sequenc
e
template class
template
specialization
lambda
functor
<algorithm>
iterator
category
pair
tuple
template ...
vector<bool>
bitset
array
(un)ordered_)(m
ulti)map/set
priority_queue
deque list
forward_list
stack queue
char/string
literals
brace
initialization
initializer_list
vector
string
char *
iostream
<filesystem>
<locale>
<regex>

copy/move

copy/move

- Nejběžnější kombinace
  - Neškodná třída
    - Nedeklaruje žádnou copy/move metodu ani destruktor
    - Neobsahuje složky vyžadující zvláštní péči (ukazatele)
  - Třída obsahující složky vyžadující zvláštní péči
    - Překladačem generované chování (default) nevyhovuje
    - Bez podpory move (typický stav před C++11, dnes funkční, ale neoptimální)

```
T( const T & x);
T & operator=( const T & x);
~T() noexcept;
```

    - Plná podpora copy/move

```
T( const T & x);
T( T && x) noexcept;
T & operator=( const T & x);
T & operator=( T && x) noexcept;
~T() noexcept;
```

copy/move

- Další kombinace
  - Nekopírovatelná a nepřesouvatelná třída
    - Např. dynamicky alokované živé objekty v simulacích

```
T( const T & x) = delete;

T & operator=( const T & x) = delete;
```

    - delete zakazuje generování překladačem
    - Destruktor může ale nemusí být nutný

  - Přesouvatelná nekopírovatelná třída
    - Např. unikátní vlastník jiného objektu (std::unique_ptr< U>)

```
T( T && x) noexcept;

T & operator=( T && x) noexcept;

~T() noexcept;
```

    - Pravidla jazyka zakazují generování copy metod překladačem
    - Destruktor typicky bývá nutný

Konstruktory a spol.

- Abstraktní třída
  - Se zákazem kopírování/přesouvání

```
class T {
protected:
  T() {}
  T( const T & x) = delete;
  T & operator=( const T & x) = delete;
public:
  virtual ~T() noexcept {}
};
```

Konstruktory a spol.

- Abstraktní třída
  - S podporou klonování

```
class T {
protected:
  T() {}
  T( const T & x) = default;
  T & operator=( const T & x) = delete;
public:
  virtual ~T() noexcept {}
  virtual T * clone() const = 0;
};
```

- Důsledky přítomnosti datových položek
  - Číselné typy
    - Vyžadují explicitní inicializaci, destrukce není nutná
    - Kopírování/přesouvání je bez problémů
  - Struktury/třídy
    - Nemají-li vlastní copy/move operace, chovají se stejně, jako kdyby jejich součásti byly přítomny přímo
    - Mají-li dobře udělané vlastní copy/move, obvykle nezpůsobují problémy
      - Vyžadují ošetření, pokud má být semantika vnější třídy jiná (např. chytré ukazatele ve třídách s hodnotovou semantikou)
  - Kontejnery a řetězce
    - Kontejnery mají vlastní copy/move operace
    - Chovají se stejně, jako kdyby elementy kontejneru byly přítomny přímo
      - Kontejner je však automaticky inicializován jako prázdný - není třeba inicializovat jeho prvky

- Důsledky přítomnosti datových položek - odkazy bez odpovědnosti vlastníka
  - Reference (U&)
    - Vyžadují explicitní inicializaci, destrukce není nutná
    - Copy/move konstruktory jsou bez problémů
    - Copy/move operator= je znemožněn
  - Ukazatele (U*) bez (spolu-)vlastnické semantiky
    - Dealokaci řeší někdo jiný
    - Vyžadují explicitní inicializaci, destrukce není nutná
    - Kopírování/přesouvání je bez problémů

Konstruktory a spol.

- POD: Plain-Old-Data
  - Položky jsou veřejné
  - Inicializace je v zodpovědnosti uživatele

```
class T {
public:
  std::string x_;
};
```

  - Často se používá struct

```
struct T {
  std::string x_;
};
```

Konstruktory a spol.

- Všechny položky jsou neškodné
  - Položky mají svoje konstruktory
  - Třída nemusí mít žádný konstruktor

```
class T {
public:
  // ...
  const std::string & get_x() const { return x_; }
  void set_x( const std::string & s) { x_ = s; }
private:
  std::string x_;
};
```

Konstruktory a spol.

- Všechny položky jsou neškodné
  - Položky mají svoje konstruktory
  - Konstruktor se hodí pro pohodlnou inicializaci
    - V takovém případě je (většinou) nutný i konstruktor bez parametrů
    - Konstruktory s jedním parametrem označeny explicit

```
class T {
public:
  T() {}
  explicit T( const std::string & s) : x_( s) {}
  T( const std::string & s, const std::string & t)
    : x_( s), y_( t)
  {}
  // ... metody ...
private:
  std::string x_, y_;
};
```

Konstruktory a spol.

- Některé položky jsou mírně nebezpečné
  - Některé položky nemají vhodné konstruktory
    - Číselné typy včetně bool, char,
    - Ukazatele bez vlastnické semantiky
  - Konstruktor je nutný pro inicializaci
    - V takovém případě je (většinou) nutný i konstruktor bez parametrů
    - Konstruktory s jedním parametrem označeny explicit

```
class T {
public:
  T() : x_( 0), y_( nullptr) {}
  explicit T( int s) : x_( s), y_( nullptr) {}
  T( int s, U * t)
    : x_( s), y_( t)
  {}
  // ... metody ...
private:
  int x_; U * y_;
};
```

- Důsledky přítomnosti datových položek - chytré odkazy
  - std::unique_ptr<U>
    - Explicitní inicializace není nutná (automaticky nullptr)
    - Explicitní destrukce není nutná (automaticky dealokuje připojený objekt)
    - Copy operace nejsou možné
      - Jsou-li vyžadovány, musejí být implementovány kopírováním obsahu
    - Move operace jsou bezproblémové
  - std::shared_ptr<U>
    - Explicitní inicializace není nutná (automaticky nullptr)
    - Explicitní destrukce není nutná
    - Copy operace fungují, ale mají semantiku sdílení
      - Mají-li mít semantiku kopírování, je nutné upravit ostatní metody třídy
        - všechny modifikující metody si musejí vytvořit privátní kopii připojeného objektu
    - Move operace jsou bezproblémové

Konstruktory a spol.

- Použití unique_ptr
  - Třída se zakázaným kopírováním
    - Ale schopná přesunu

```
class T {
public:
  T() : p_( new Data) {}
private:
  std::unique_ptr< Data> p_;
};
```

Konstruktory a spol.

- Použití unique_ptr
  - Třída s povoleným kopírováním

```
class T {
public:
  T() : p_( new Data) {}
  T( const T & x) : p_( new Data( * x.p_)) {}
  T( T && x) = default;
  T & operator=( const T & x) { return operator=( T( x));}
  T & operator=( T && x) = default;
private:
  std::unique_ptr< Data> p_;
};
```

- Důsledky přítomnosti datových položek - vlastnické odkazy
  - Ukazatele (U*) se semantikou vlastníka
    - Naše třída musí řešit dealokaci připojeného objektu
    - Vyžadují explicitní inicializaci (alokace nebo vynulování)
    - Destrukce je nutná (dealokace)
    - Kopírování musí alokovat nový připojený objekt a kopírovat jeho obsah
    - Přesouvání musí vynulovat odkazy ve zdrojové třídě
    - Copy/move operator= musí navíc uklidit původní obsah
  - Ukazatele (U*) se semantikou spoluvlastníka
    - Naše třída musí řešit počítání odkazů a dealokaci připojeného objektu
    - Vyžadují explicitní inicializaci (alokace nebo vynulování)
    - Destrukce je nutná (odečtení odkazu, dealokace)
    - Kopírování musí aktualizovat počet odkazů
    - Přesouvání musí vynulovat odkazy ve zdrojové třídě
    - Copy/move operator= musí navíc uklidit původní obsah

Konstruktory a spol.

- Některé položky jsou hodně nebezpečné
  - Syrové ukazatele s vlastnickou semantikou
  - Je nutný copy/move constructor/operator= a destruktor
    - Je nutný i jiný konstruktor, např. bez parametrů

```
class T {
public:
  T() : p_( new Data) {}
  T( const T & x) : p_( new Data( * x.p_)) {}
  T( T && x) : p_( x.p_) { x.p_ = 0; }
  T & operator=( const T & x) { T tmp( x); swap( tmp); return * this;}
  T & operator=( T && x)
    { T tmp( std::move( x)); swap( tmp); return * this;}
  ~T() { delete p_; }
  void swap( T & y) { std::swap( p_, y.p_); }
private:
  Data * p_;
};
```

Pointer vs. value

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

Raw pointers (C++)

```
class T {
public:
  int a;
};

void f( T * z)
{
  z->a = 3;
}

void g()
{
  T * x = new T;
    // allocation

  x->a = 1;

  T * y = x;
    // second pointer

  y->a = 2;
    // x->a == 2

  f( x);
    // x->a == 3

  delete x;
    // manual deallocation
}
```

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

Smart pointers (C++)

```
class T {
public:
  int a;
};

void f( T * z)
{
  z->a = 3;
}

void g()
{
  std::shared_ptr< T> x =
    std::make_shared< T>();
    // allocation

  x->a = 1;

  std::shared_ptr< T> y = x;
    // second pointer

  y->a = 2;
    // x->a == 2

  f( x);
    // x->a == 3

  // automatic deallocation
  // when pointers are destructed
}
```

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

References (C++)

```
class T {
public:
  int a;
};

void f( T & z)
{
  z.a = 3;
}

void g()
{
  T x;     // automatic storage (stack)

  x.a = 1;

  T & y = x;
    // a reference to the stack object

  y.a = 2;
    // x.a == 2

  f( x);
    // x.a == 3

  // x is destructed on exit
}
```

C#/Java vs. C++

Value types (C#)

```
struct T {
  int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x;
      // creation

    x.a = 1;

    T y = x;
      // a copy

    y.a = 2;
      // x.a == 1

    f( x);
      // x.a == 1

      // destruction on exit
  }
}
```

Values (C++)

```
class T {
public:
  int a;
};

void f( T z)
{
  z.a = 3;
}

void g()
{
  T x;
    // creation

  x.a = 1;

  T y = x;
    // a copy

  y.a = 2;
    // x.a == 1

  f( x);
    // x.a == 1

    // destruction on exit
}
```

C#/Java vs. C++

Passing value types by reference (C#)

```
struct T {
  int a;
}

class test {
  static void f( ref T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x;
      // creation

    x.a = 1;

    f( ref x);
      // x.a == 3
  }
}
```

Passing by lvalue reference (C++)

```
class T {
public:
  int a;
};

void f( T & z)
{
  z.a = 3;
}

void g()
{
  T x;

  x.a = 1;

  f( x);
    // x.a == 3
}
```

C#/Java vs. C++

Passing reference types by reference (C#)

```
class T {
  public int a;
}

class test {
  static void f( ref T z)
  {
    z = new T();
      // allocation of another object
  }

  static void g()
  {
    T x = new T();
      // allocation

    f( ref x);
      // x is now a different object

    // deallocation later by GC
  }
}
```

Passing smart pointers by reference (C++)

```
class T {
public:
  int a;
};

void f( std::unique_ptr<T> & z)
{
  z = new T;
    // allocation of another object
    // deallocation of the old object
}

void g()
{
  std::unique_ptr< T> x = new T;
    // allocation

  f( x);
    // *x is now a different object

  // deallocation by destruction of x
}
```

Je lepší C++ nebo Java/C#?

Je lepší C++ nebo Java/C#?

Špatná otázka

Co programovat v C++

- Pro které oblasti je C++ lepší než Java/C#?
  - Důraz na výkon
    - C++ umožňuje programovat způsobem, který neubírá na výkonu
    - Když budete programovat v C++ stejným stylem jako v Java/C#, dostanete přibližně stejný výkon
  - Spolupráce s hardware
    - C++ nechystá na programátora nepříjemná překvapení (garbage collection etc.)
    - Embedded assembler, spojování s jinými jazyky
  - Spolupráce s OS
    - Všechny významné OS mají v C jádro a tudíž i rozhraní OS
    - Většina systémových aplikací je v C nebo C++
    - Nativní knihovny jazyků Java/C# jsou implementovány v C/C++
  - Generické programování
    - Mechanismus šablon v C++ je silnější než v Java/C#
    - Způsob implementace šablon v C++ neubírá na výkonu

Co programovat v C++

- Proč je Java/C# pomalejší?
  - Java/C# nutí k dynamické alokaci vždy a všude
    - V C++ lze dynamickou alokaci používat pouze v nezbytných případech
      - datové struktury proměnlivé velikosti
      - polymorfní datové struktury
      - objekty s nepravidelnou dobou života
  - Garbage collection je pomalejší než explicitní dealokace (delete)
    - GC dealokuje pozdě - problémy s využitím cache
    - GC je ale rychlejší než chytré ukazatele (shared_ptr)
      - Programy psané v C++ stylem Java/C# jsou pomalejší než originál
  - Chybí pokročilé metody optimalizace v překladačích
    - Vektorizace, transformace cyklů, ...
    - Překladače nemají čas (JIT), jejich autoři motivaci
    - Existují i situace, kdy je Java/C# rychlejší
      - Překladače Javy/C# mají jednodušší úlohu při analýze kódu
- Je C++ pomalejší než C?
  - Ne, pokud v něm neprogramujete jako v Javě/C#

Co neprogramovat v C++

Co raději **ne**programovat v C++

- Interaktivní aplikace s GUI
  - C++ nemá standardizované rozhraní na GUI
  - Nativní rozhraní GUI v OS je většinou archaické C
    - Přímé použití je obtížné a nebezpečné
  - Knihovny pro GUI jsou archaické, nepřenositelné nebo obojí
    - Qt, GTK+, wxWidgets...
  - Garbage Collection při programování GUI citelně chybí
- Aplikace skládané z mnoha cizích součástí
  - V C++ neexistuje široce uznávaný styl psaní komponent
  - Standard C++ nedostatečně zprostředkovává služby OS, Internetu atd.
    - Situace v C++11 a C++14 je však daleko lepší než dříve
  - Cizí knihovny obvykle doplňují chybějící části vlastní tvorbou
    - Různé implementace chybějících částí mohou být v konfliktu
- Pokud je ale zároveň zapotřebí výkon, nic moc jiného než C++ nezbývá

Proč C++

- Proč (stále ještě) učíme C++?
  - Většina řadových programátorů v C++ programovat nebude
  - MFF chce vychovávat elitu
    - Programování OS, databází, překladačů
    - Vědecké výpočty vyžadující výkon
    - Hry, robotika, vestavěné systémy ...
    - Údržba rozsáhlých a historických softwarových systémů
  - Porozumíte-li tomu, jak funguje C++, budete lépe rozumět
    - jiným programovacím jazykům
    - architektuře počítačů a operačních systémů
    - překladačům
  - Zvládnutí C++ je odznakem zdatnosti matfyzáka

Setter bez konverze – různé implementace

Tradiční před C++11

```
class packet {
public:
  void set_contents(const std::string & s)
  { data_ = s; }
private:
  std::string data_;
};
```

- Pokud je skutečným parametrem L-value, lepší řešení neexistuje

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** z *my_string* do *data_*
  - tato operace může recyklovat původní prostor alokovaný v proměnné *data_*

- Pokud je skutečným parametrem R-value, dochází ke zbytečnému kopírování

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***copy*** do *data_*

Zkrácené v C++11

```
class packet {
public:
  void set_contents(std::string s)
  { data_ = std::move( s); }
private:
  std::string data_;
};
```

- Pokud je skutečným parametrem L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** z *my_string* do *s*
  - ***move*** z *s* do *data*
  - recyklace původního prostoru není možná

- Pokud je skutečným parametrem R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***move*** do *s*
  - ***move*** z *s* do *data_*

Úplné v C++11

```
class packet {
public:
  void set_contents(const std::string & s)
  { data_ = s; }
  void set_contents(std::string && s)
  { data_ = std::move( s); }
private:
  std::string data_;
};
```

- Pokud je skutečným parametrem L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** z *my_string* do *data_*
    - tato operace může recyklovat původní prostor alokovaný v proměnné *data_*

- Pokud je skutečným parametrem R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***move*** do *data_*
    - zahrnuje dealokaci původního prostoru

Setter bez konverze – různé implementace

Úplné v C++11

```
class packet {
public:
  void set_contents(const std::string & s)
  { data_ = s; }
  void set_contents(std::string && s)
  { data_ = std::move( s); }
private:
  std::string data_;
};
```

- Pokud je skutečným parametrem L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** z *my_string* do *data_*
    - tato operace může recyklovat původní prostor alokovaný v proměnné *data_*

- Pokud je skutečným parametrem R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***move*** do *data_*
    - zahrnuje dealokaci původního prostoru

Generické v C++11

```
class packet {
public:
  template< typename X>
  void set_contents(X && s)
  { data_ = std::forward< X>( s); }
private:
  std::string data_;
};
```

- Pokud je skutečný parametr typu *std::string*, chování je shodné s variantou dvou funkcí
  - *std::forward* je podmíněná varianta *std::move* pro univerzální reference
- Pokud je skutečný parametr jiného typu (např. *char[14]*)

```
p->set_contents( "Hello, world!");
```

  - ke konverzi dochází uvnitř funkce
  - může existovat speciální varianta *operator=* pro tento typ
    - ta může recyklovat původní prostor
  - v opačném případě se provede
    - konverze, zahrnující alokaci nového prostoru
    - *move* do *data_*

Dočasné zpřístupnění objektu

```
class packet {
public:
  void set_contents( std::string s)
  { data_ = std::move( s); }
  const std::string & get_contents() const
  { return data_; }
private:
  std::string data_;
};
```

- Třída *packet* obsahuje data ve formátu srozumitelném pro její uživatele
  - Zápis a čtení dat nevyžaduje konverzi
- *set_contents* přijímá parametr *p* hodnotou
  - pokud je skutečným argumentem R-value, bude do parametru *s* přesouvána, v opačném případě kopírována
  - funkce přesouvá obsah parametru do položky *data_*
    - to by nebylo možné pří předávání jako *const std::string &*

```
void send_hello()
{
  auto p = std::make_unique< packet>();
  p->set_contents( "Hello, world!");
  ch.send( std::move( p));
}
```

```
void dump_channel()
{
  while ( ! ch.empty() )
  {
    auto m = ch.receive();
    std::cout << m->get_contents();
    // the packet is deallocated here
  }
}
```

- *get_contents* vrací odkaz na vniřní data
  - *const* brání modifikaci
- Jak dlouho tato reference platí?
  - Až do modifikace/destrukce objektu *packet*
  - Příjemce je unikátním vlastníkem, takže tyto události má pod kontrolou
  - Reference platí přinejmenším v rámci jednoho příkazu
    - Pokud v něm není jiná akce s tímtéž objektem *packet*
    - Dvě akce s tímtéž objektem v jednom příkazu obvykle způsobují i další problémy (nejisté pořadí apod.)