Programming in C++

Course credits and grading

Nov	Homework assignment #1	0 .. 15 pts	Specific per lab group
Nov .. Dec	Select programming project theme	Mandatory	Contact your lab teacher
Dec .. Jan	Homework assignment #2	0 .. 25 pts	Specific per lab group
Jan .. Feb (Apr)	Practical exam in lab	0 .. 60 pts	Common, enroll in SIS
(on demand)	Optional oral exam	-10 .. +10 pts	Contact your examiner or lecturer
until May	Deliver the programming project	Mandatory	Contact your lab teacher

- Lab credit = 50 pts + programming project
  - Not required for practical exam
- Grading = homework assignments + practical exam (+ oral exam)
  - 3 = 60 pts
  - 2 = 75 pts
  - 1 = 90 pts

Course credits and grading

Homework #1	0..15 points	Delay penalty: -5 points per each week (even if partial)
Homework #2	0..25 points	Delay penalty: -10 points per each week (even if partial)
Practical part	0..60 points	50 points for a completely functional solution, +/- 10 points for the quality of source code
Optional oral part	-10..+10 points	At least 50 points from the previous parts required for admission.

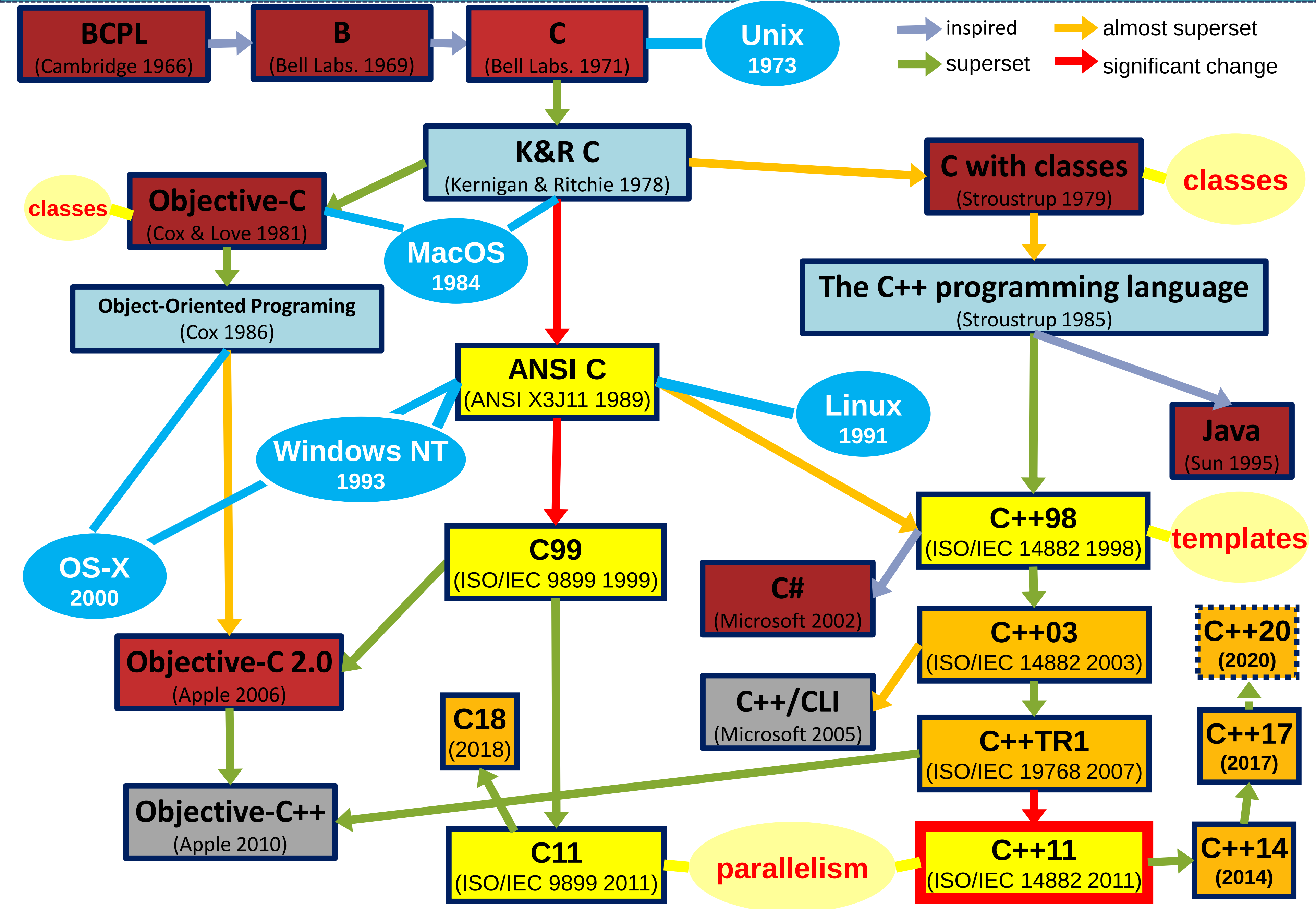
- Lab credit = 50 pts + programming project
  - Not required for practical exam
- Grading = homework assignments + practical exam (+ oral exam)
  - 3 = 60 pts
  - 2 = 75 pts
  - 1 = 90 pts

Course credits

- Conditions may be individually adjusted:
  contact your lab teacher during October
  - Erasmus students may need dates and deadlines sooner
- If you failed previous year (or before)
  - Contact now your (new) lab teacher
- If you fail this year
  - Your points may be retained if you are reasonably succesful
  - Your success in programming project will be retained

History and Literature

History of C++



inspired
superset
almost superset
significant change
BCPL
(Cambridge 1966)
B
(Bell Labs. 1969)
C
(Bell Labs. 1971)
Unix
1973
K&R C
(Kernigan & Ritchie 1978)
C with classes
(Stroustrup 1979)
classes
Objective-C
(Cox & Love 1981)
classes
MacOS
1984
The C++ programming language
(Stroustrup 1985)
Object-Oriented Programing
(Cox 1986)
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)

Books

- http://stackoverflow.com/questions/388242/the-definitive-c-book-guide-and-list
- Be sure that you have (at least) 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

The C++ Programming Language


Java Setup - Progress
Java
ORACLE
Status: Installing Java
ATMs, Smartcards, POS Terminals, Blu-ray Players, PCs
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/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, 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
  - Replaced by smart pointers 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
- 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

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
```

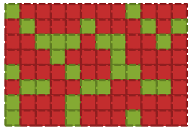
- Less readable assembly code

```
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
```

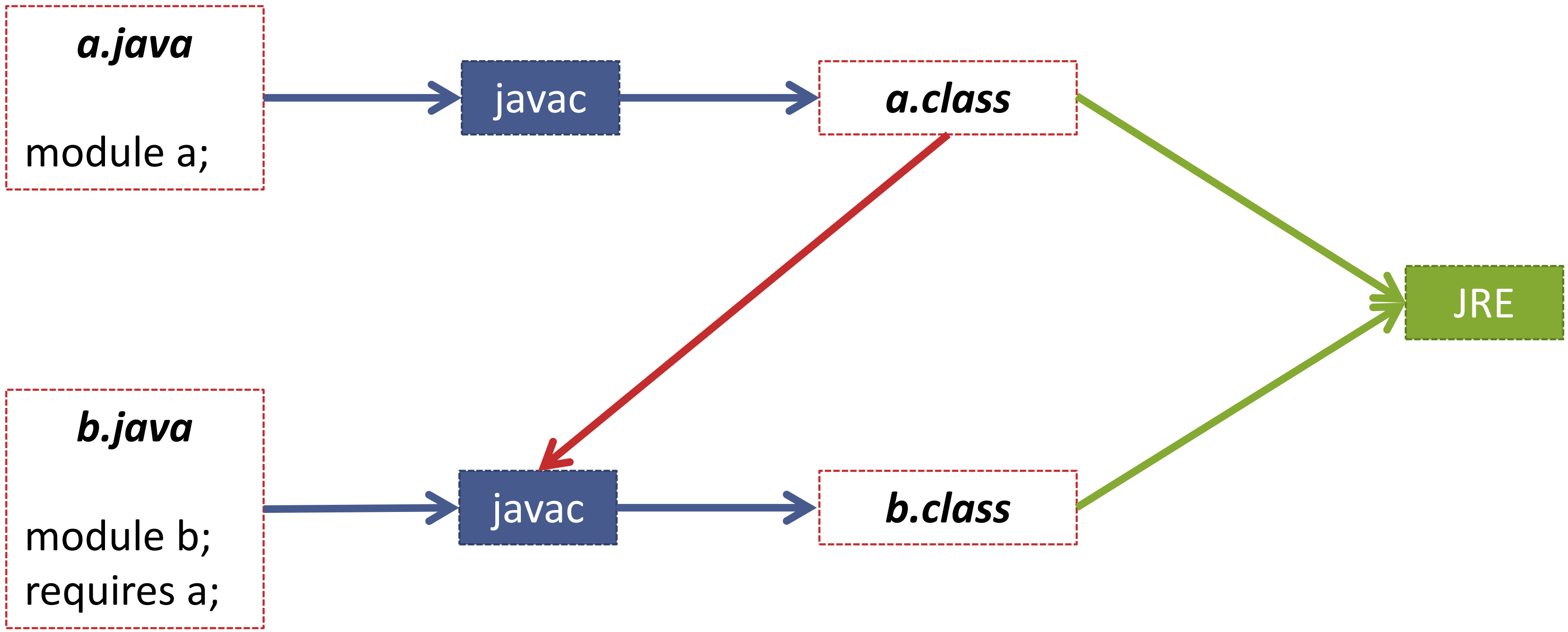
- “Machine readable” binary code



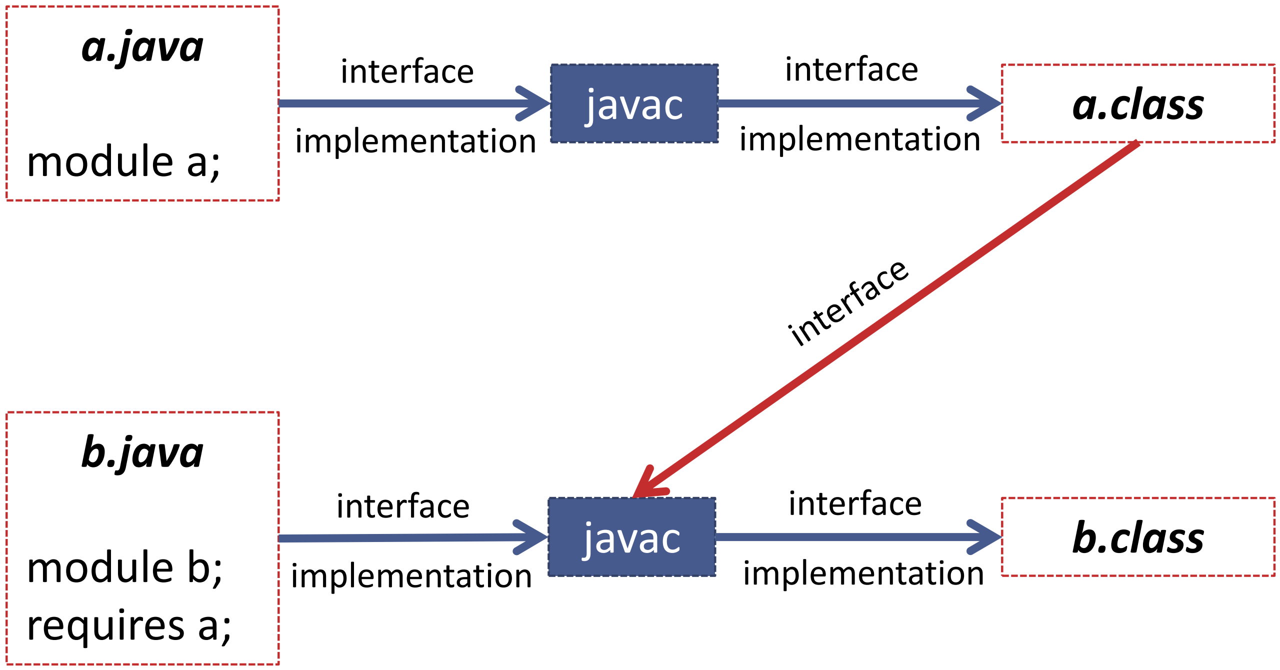
- The role of the compiler
  - Allocate space for the variables

```
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

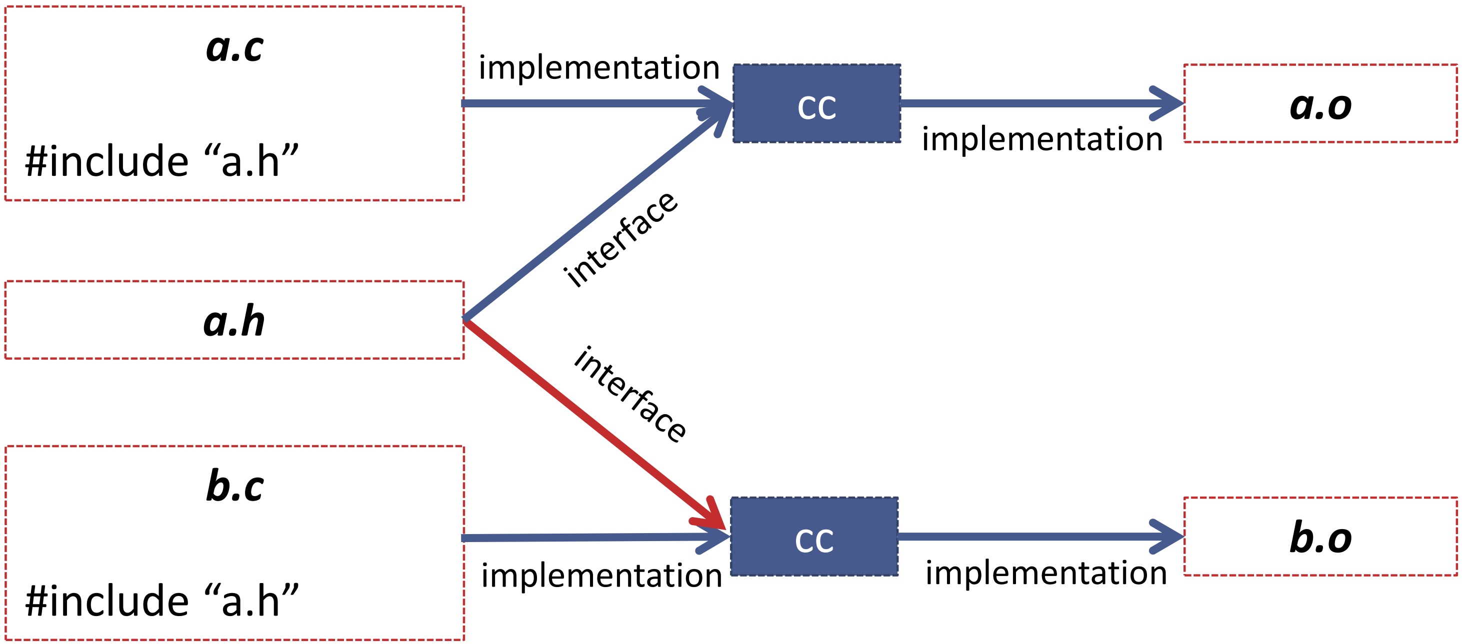


- Most modern languages compile source code into binary packages
  - These packages are also read by the compiler when referenced
- But not in C/C++ (yet)
  - C++20 will have modules and module interfaces, more complex than in java



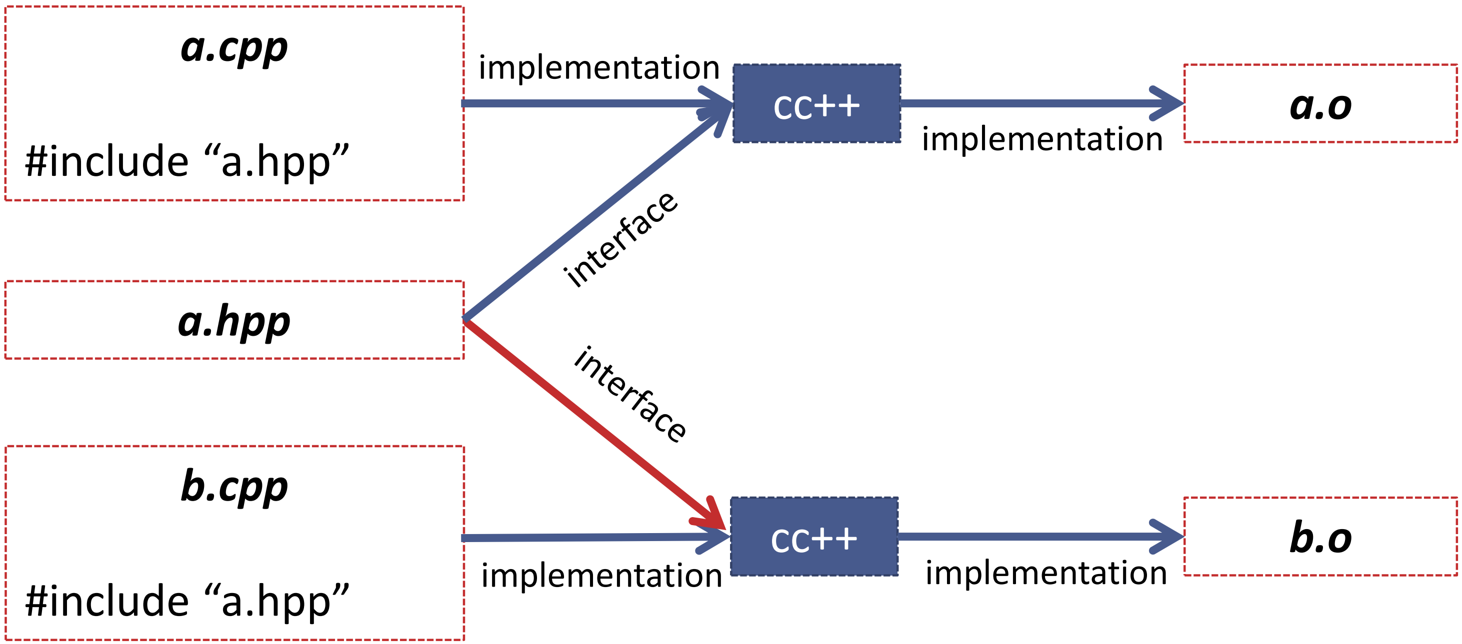
- Why not in C/C++? There are disadvantages:
  - When anything inside a.java changes, new timestamp of a.class induces recompilation of b.java
    - Even if the change is not in the public interface
  - How do you handle cyclic references?

Compilation in C



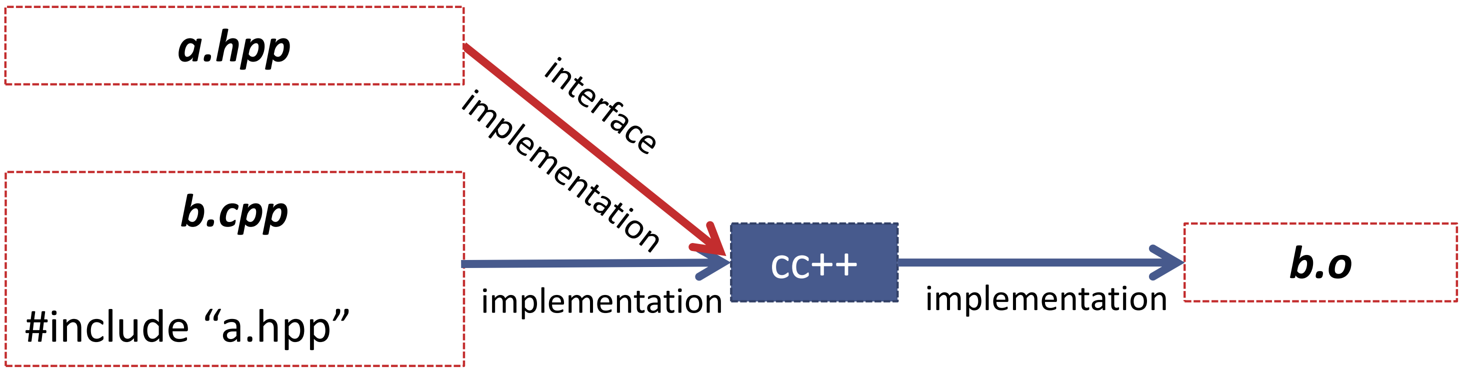
- In C, the situation was simple
  - Interface = function headers in „header files“
    - Typically small
  - Implementation = function bodies in “C files”
    - Change of a.c does not require recompilation of b.c

Compilation in C++



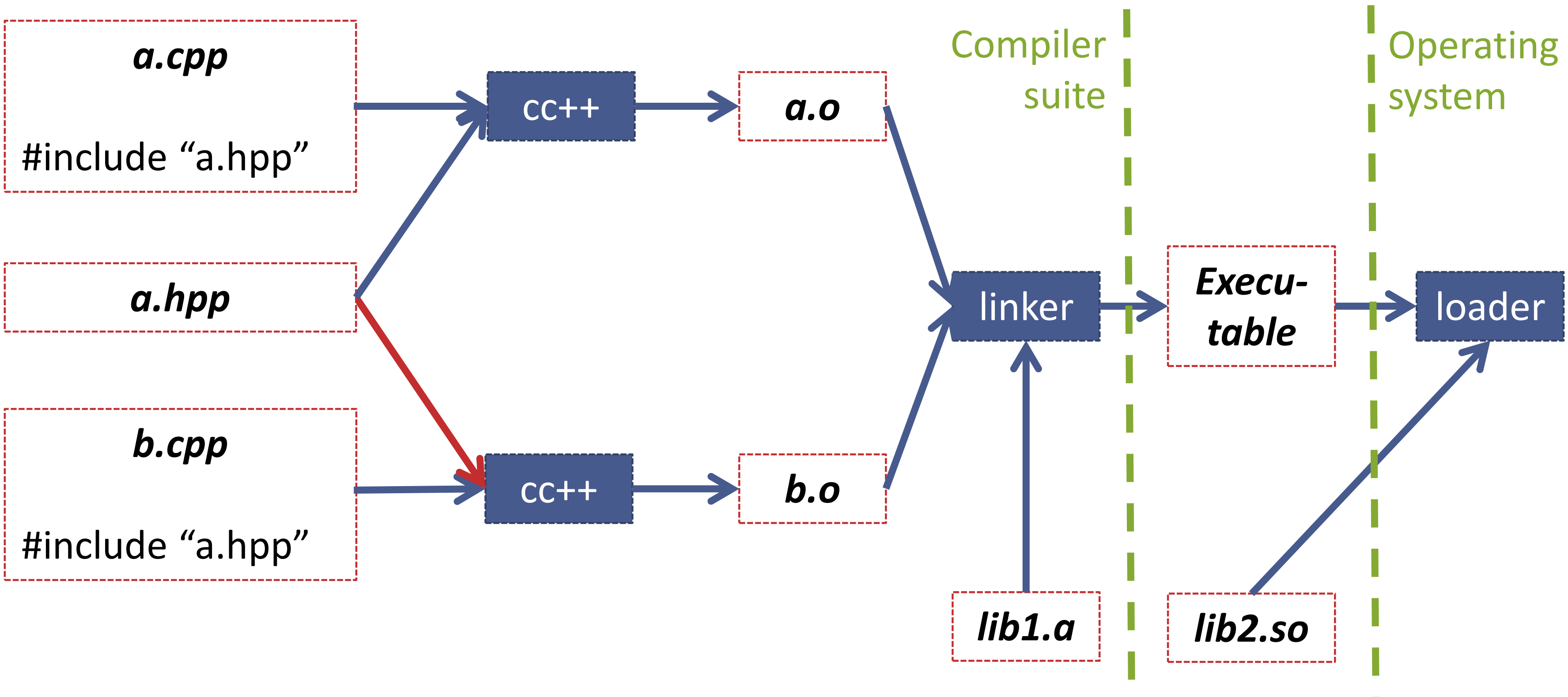
- In modern C++, the separate compilation is no longer advantage
  - Interface (classes etc.) is often larger than implementation (function bodies)
  - Changes often affect the interface, not the body
- The purely textual behavior of #include is anachronism

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!

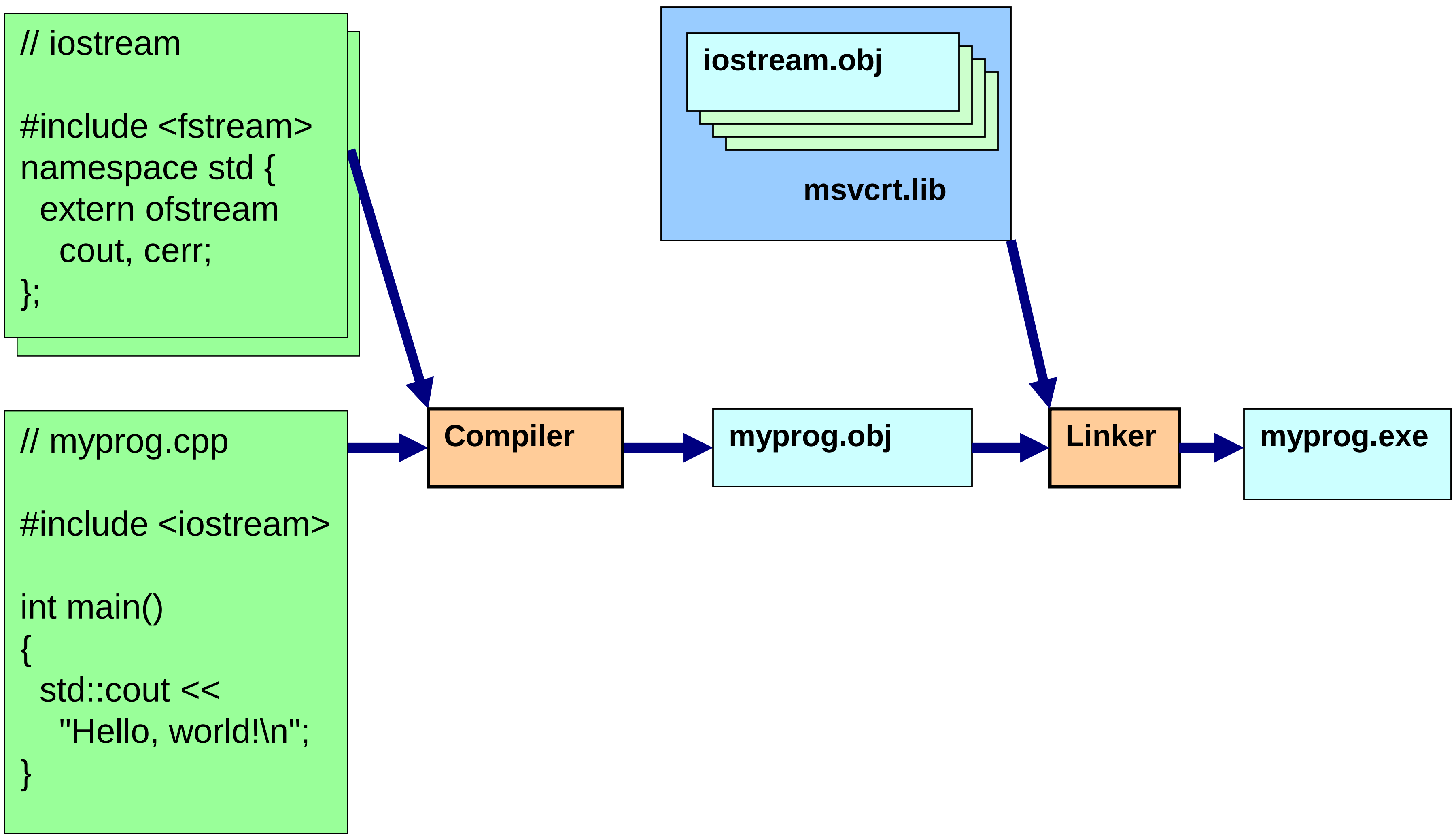
```
// world.hpp
#ifndef WORLD_HPP_
#define WORLD_HPP_

#include <vector>
#include <string>

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

#endif
```

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

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

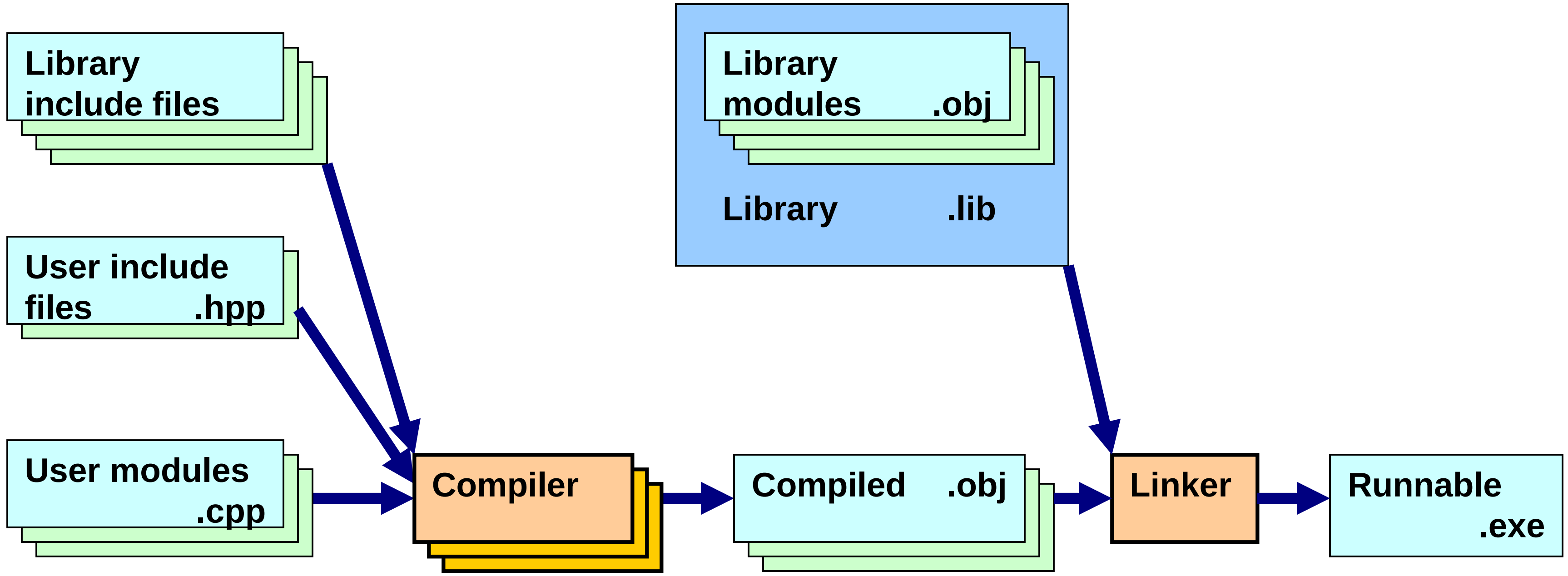
```
// world.cpp
#include "world.hpp"
#include <iostream>

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

Compilation and linking

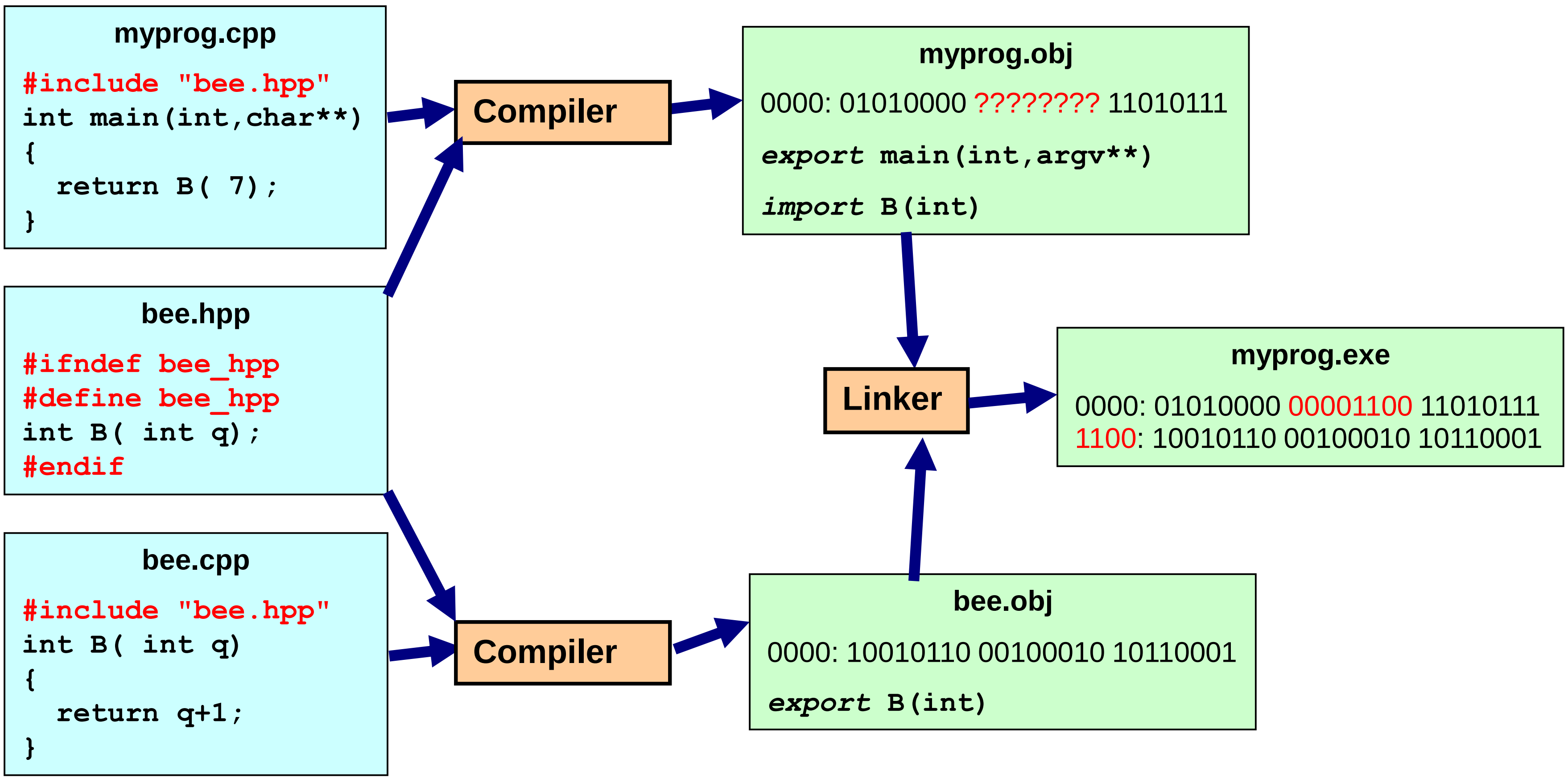
Single-module programs - static linking



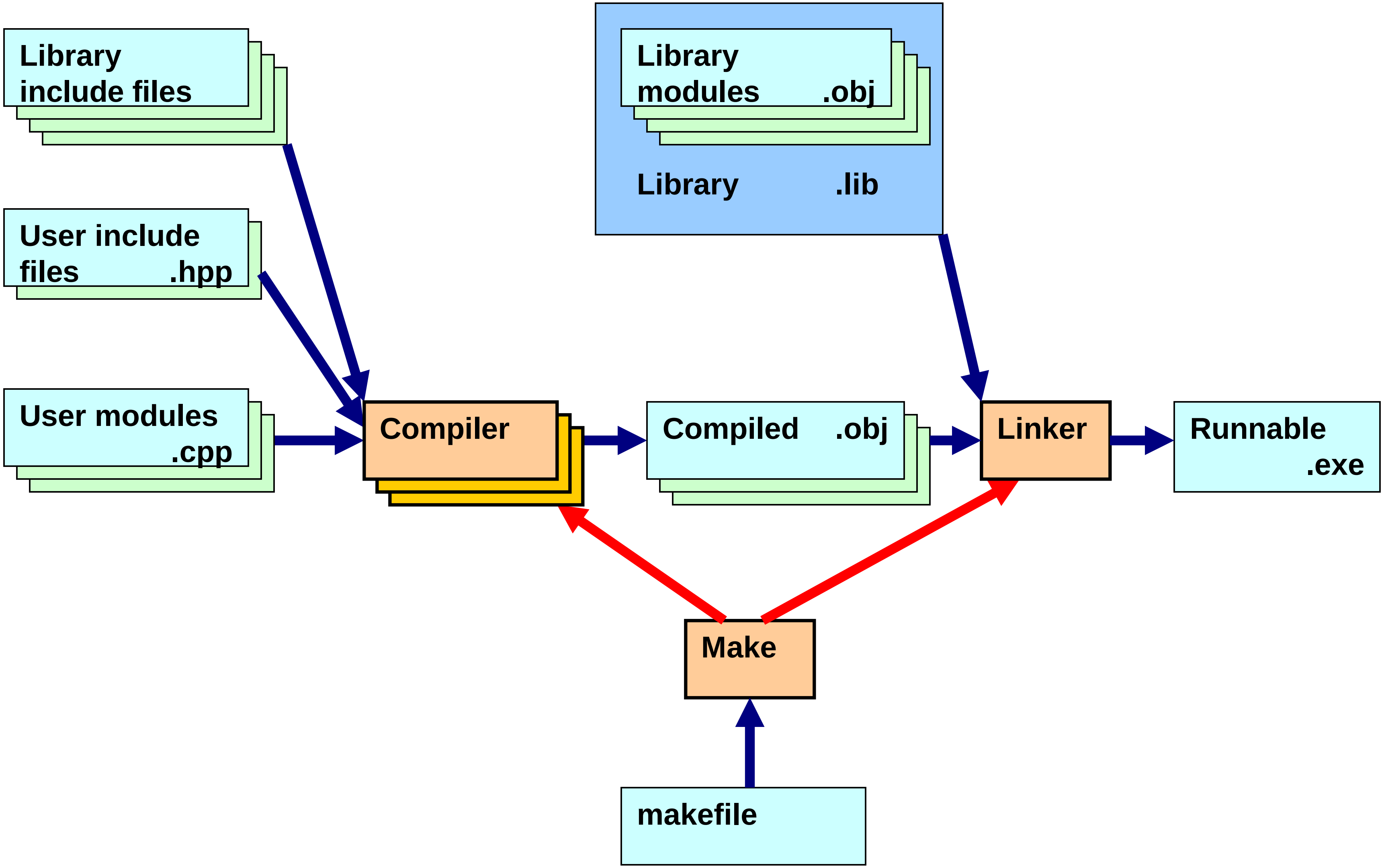
Multiple-module programs



Module interfaces and linking

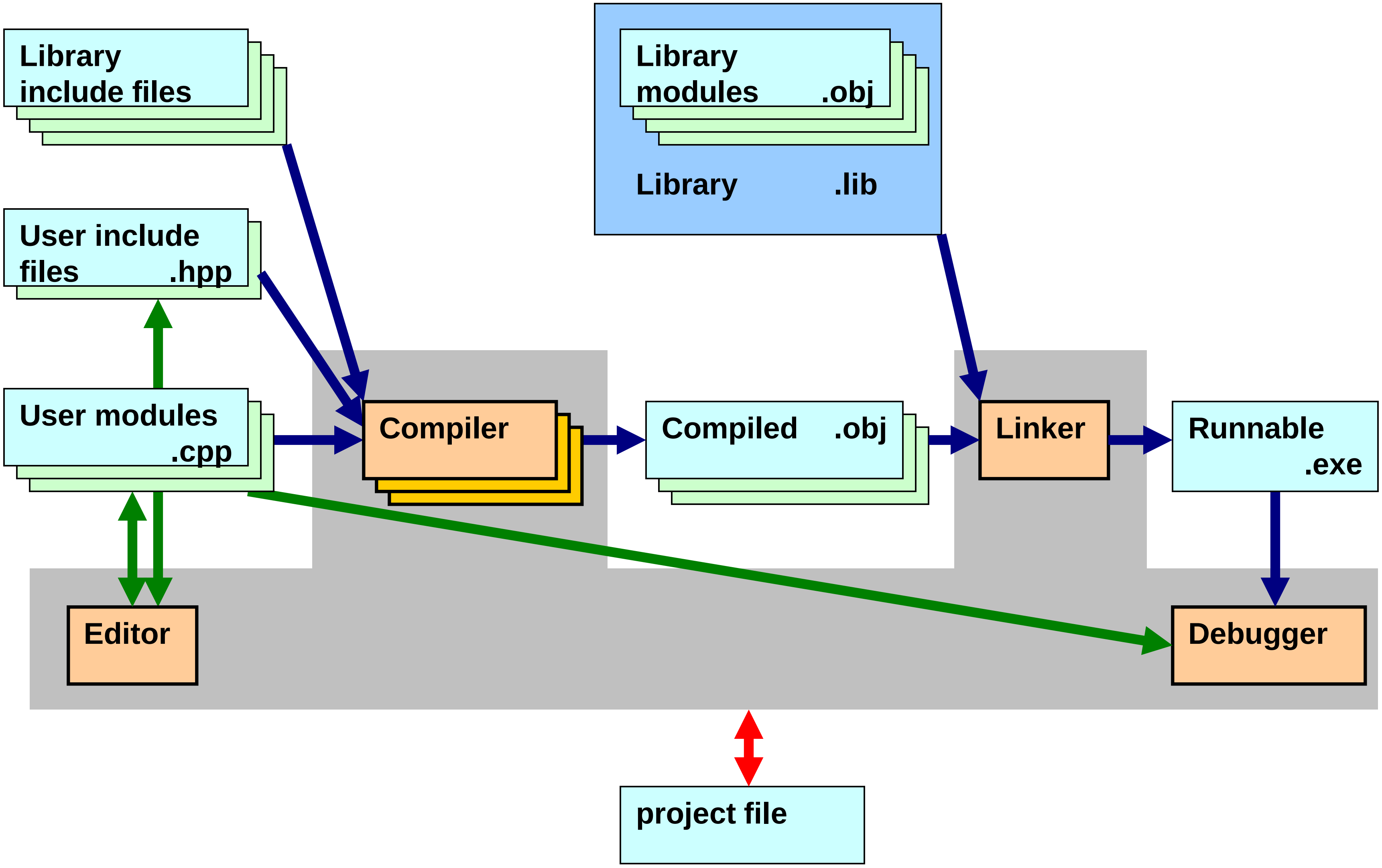


make

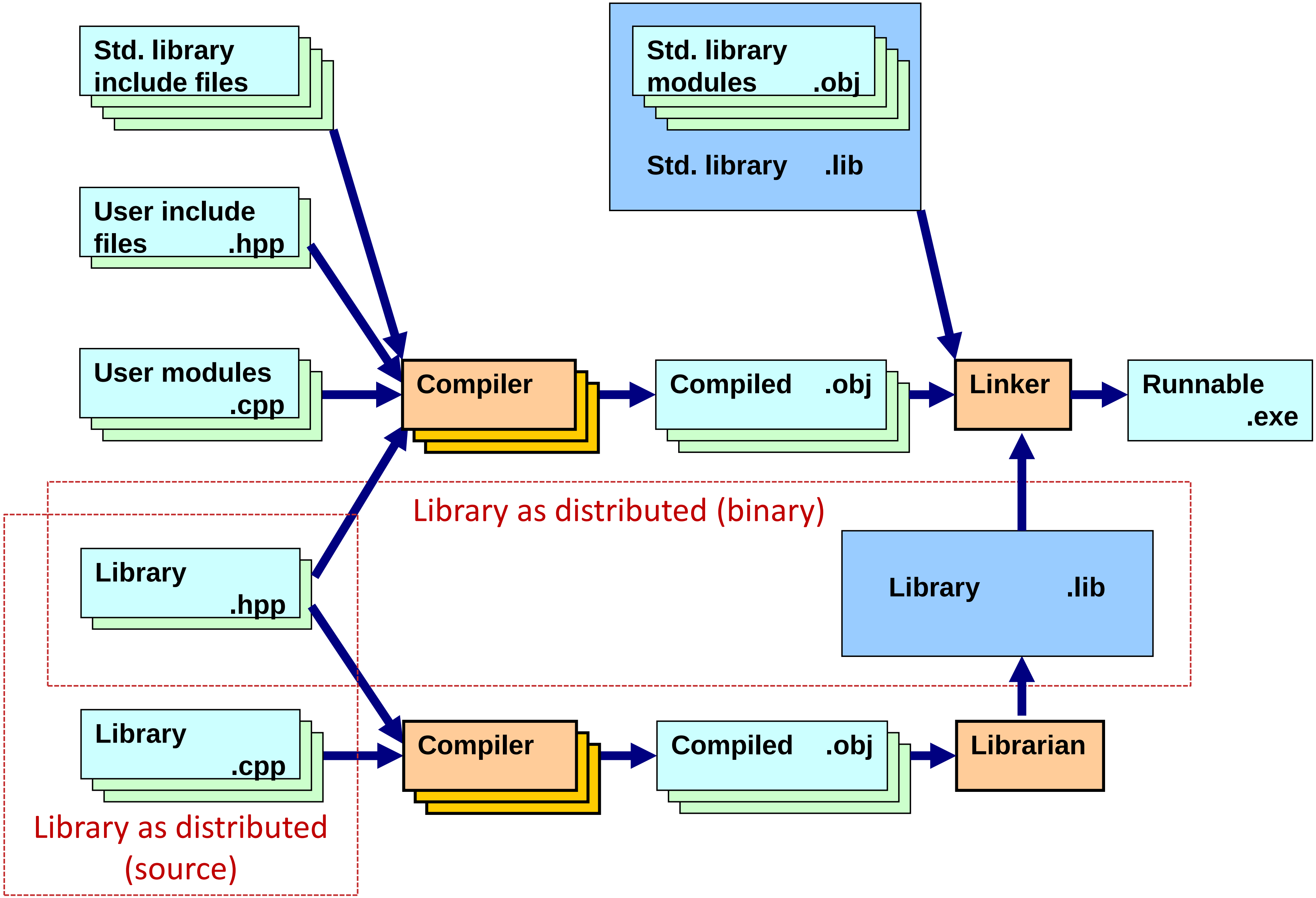


Library
include files
User include
files .hpp
User modules
.cpp
Compiler
Library
modules .obj
Library .lib
Compiled .obj
Linker
Runnable
.exe
Make
makefile

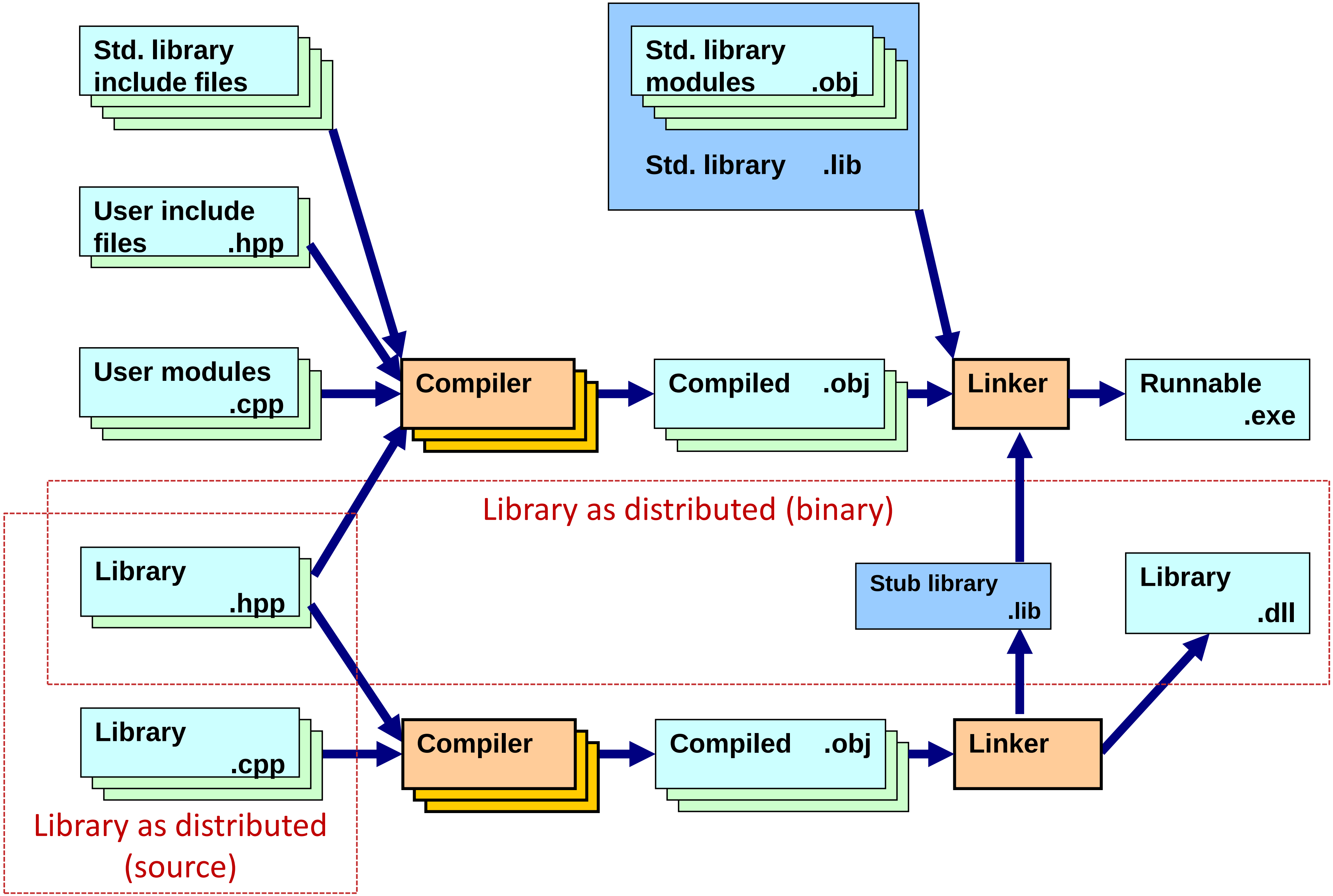
Integrated environment



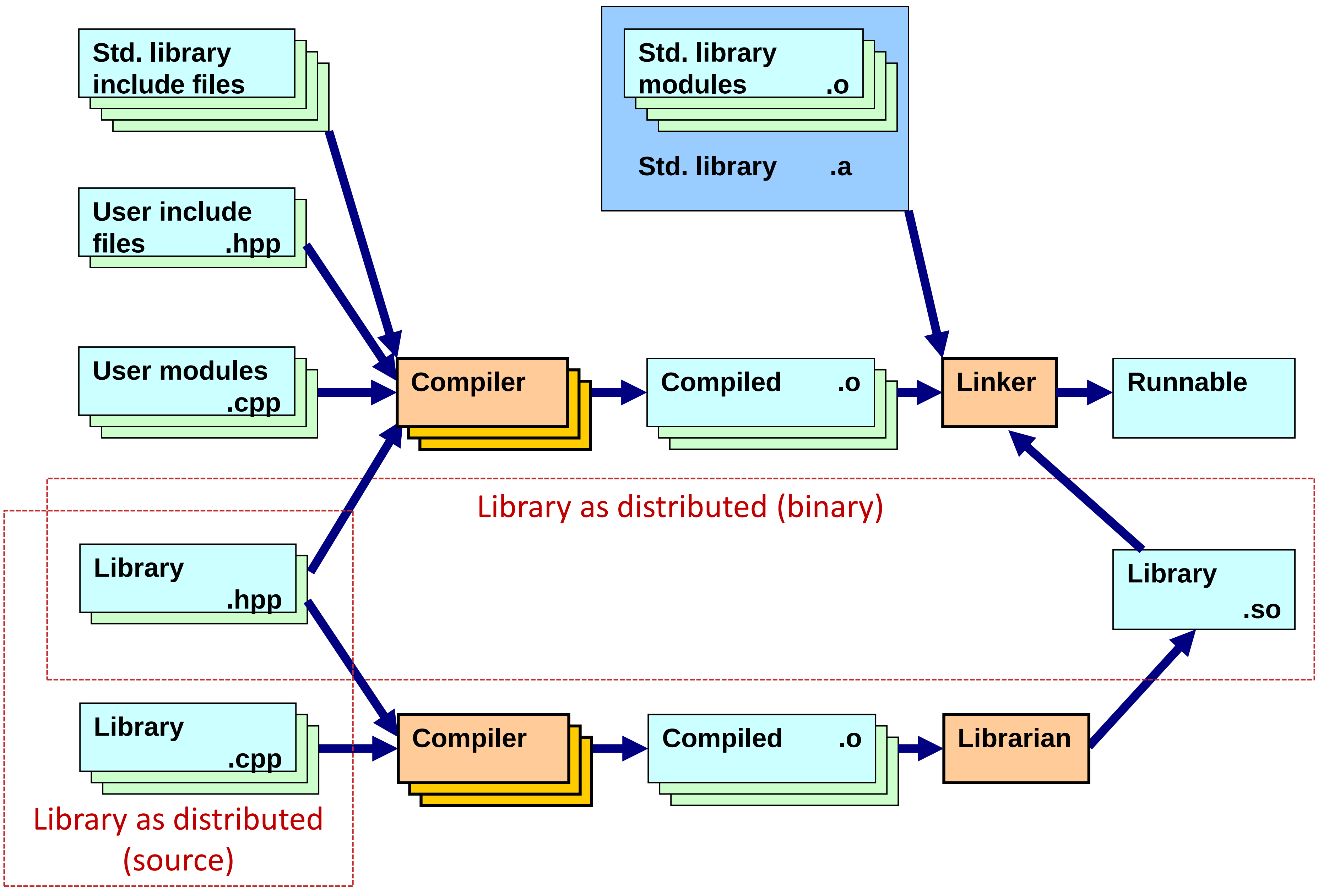
Static libraries



Dynamic libraries (Microsoft)



Dynamic libraries (Linux)



- .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

- .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

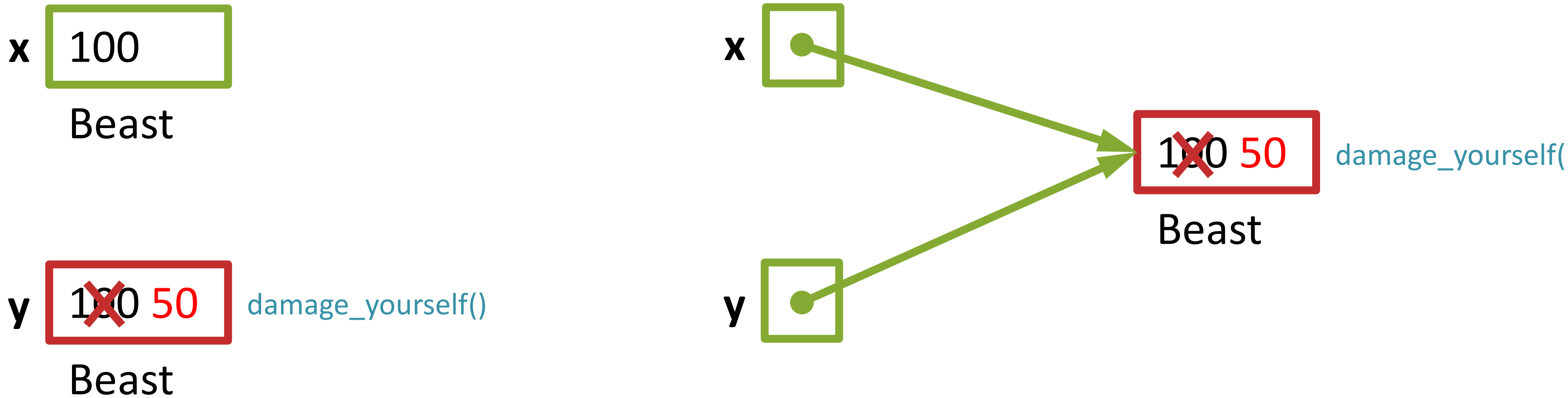
- All identifiers must be declared prior to first use
  - Compilers read the code in one pass
  - Exception: Member-function bodies are analyzed at the end of the class
    - A member function body may use other members declared later
  - Generic code involves similar but more elaborate rules
- Cyclic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

Values vs. references

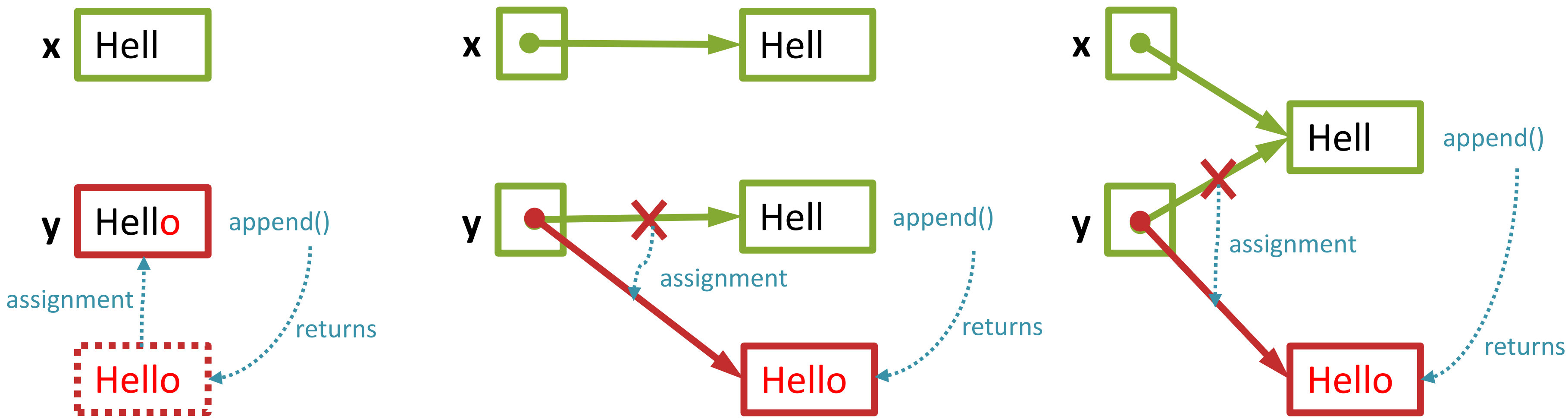
Value vs. reference types



- 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; }
```

References

Forms of pointers in C++

- References

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

  - Built in C++
  - Must be initialized to point to an object, **cannot be redirected** later
  - Syntactically identical to values when used (r.a)

- Raw pointers

`T *, const T *`

  - Built in C/C++
  - Requires special operators to access the referenced value (*p, p->a)
  - **Pointer arithmetics** allows to access adjacent values residing in arrays
  - Ownership semantics requires manual deallocation – **not recommended after C++11**

- Smart pointers

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

  - Class templates in standard C++ library
  - Operators to access the referenced value same as with raw pointers (*p, p->a)
  - **Represents ownership** - automatic deallocation on destruction of the last reference

- Iterators

`K::iterator, K::const_iterator`

  - Classes associated to every kind of container (K) in standard C++ library
  - Returned by container functions as **pointers to container elements**
  - Operators to access the referenced value same as with raw pointers (*p, p->a)
  - **Pointer arithmetics** allows to access adjacent values in the container

References in C++

- References may be used only in some contexts
  - Formal arguments of functions (almost always safe and useful)
    - Like passing by reference in other languages (but more complex)
  - Return values of functions (dangerous but sometimes necessary)
  - Local variables (sometimes useful, particularly with **auto &&**)
  - Static variables, data members of classes (limited usability, use a pointer or std::ref instead)
- References must be initialized and cannot be redirected later
  - All uses of references work as if they were the referenced object
- References have three flavors
  - Modifiable L-value reference

```
T &
```

    - The actual argument (init value) must be an L-value, i.e. a repeatedly accessible object
  - Const (L-value) reference

```
const T &
```

    - The actual argument may be anything of type T
  - R-value reference

```
T &&
```

    - The actual argument must be an R-value, i.e. a temporary object or marked with std::move()

References with function templates and auto

- Forwarding (a.k.a. universal) reference
  - As template function argument

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

  - As auto variable

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

  - May be bound to both R-values and L-values
    - Beware, there are *reference combining rules*

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

Passing arguments in C++

- Guidelines for formal **argument** types
  - If the function needs to modify the object
    - use modifiable reference: **T &**
  - otherwise, if copying of T is really cheap (numbers, complex, observer pointers)
    - pass by value: **T**
  - otherwise, if the type does not support copying
    - pass by R-value reference: **T &&**
  - [advanced] otherwise, if the function stores a copy of the object somewhere
    - if you want really fast implementation
      - implement two versions of the function, with **const T &** and **T &&**
    - simplified approach
      - pass by value: **T**
      - use std::move() when storing the object
  - otherwise
    - use const reference: **const T &**

  - These guidelines do **not** apply to return values and other contexts

Returning values from functions

- Function **return type** guidelines
  - If the function provides access to an element of a data structure (e.g. operator[])
    - and if you want to allow modification to the element
      - use modifiable L-value reference: **T &**
    - otherwise
      - use constant reference: **const T &**
    - The returned object **must survive** at least a moment **after exiting the function**

  - In all other cases
    - pass by value: **T**
    - Do not use **std::move()** in the return statement if the returned object is a local variable
      - Except when the type T does not support copying
      - The compiler will optimize the return by *copy-elision:* The local variable will be placed in the space reserved for the return value by the calling function
  - **If a function computes** or somehow constructs the returned value, **it cannot return by reference**
    - the computed value is stored only in local objects which are destroyed before the function is exited
      - the only accessible object which survives is the temporary created by the calling function to hold the returned value (the return statement initializes this temporary)

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)

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

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

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

Typical function headers

- Providing writable access to an element of a data structure

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

  - Read-only access when the data structure is read-only

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

- Returning a newly constructed value
  - With a local variable

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

  - With an anonymous temporary

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

Returning structures

- A function returning by value

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

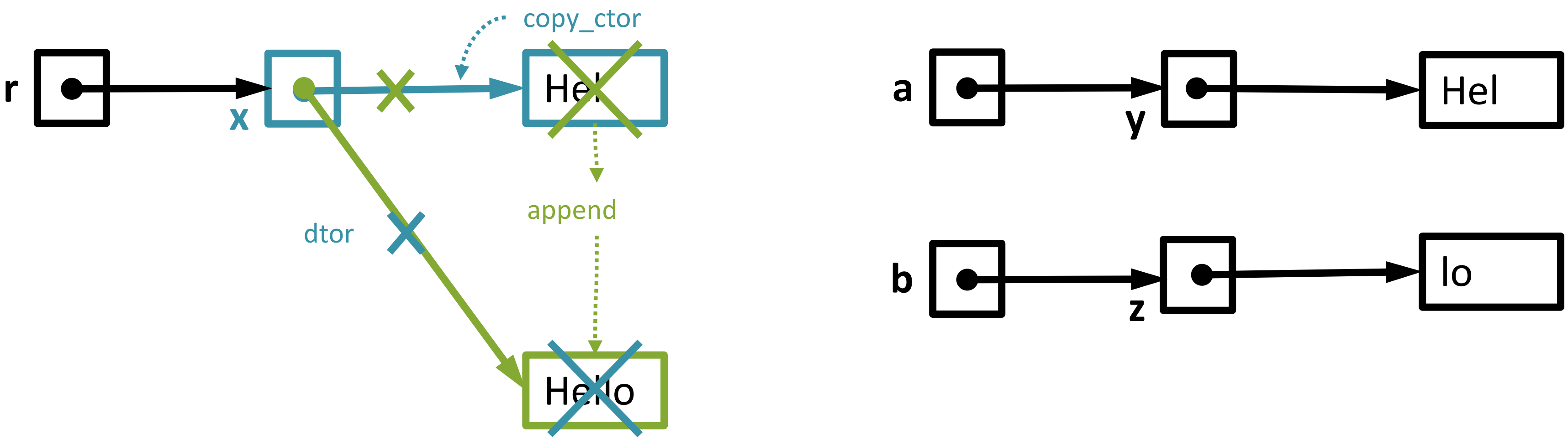
  - Used to initialize a variable

```
void f()
{ std::string x = concat( y, z); }
```

- copy/move elision
  - Mandatory behavior of compilers since C++17
  - The tmp variable is dropped, the variable from the calling function is used instead
  - This produces observable change in program behavior!
  - Translated (into a hypothetical C-like language)

```
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); x.dtor(); }
```

Function returning a value



- 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(); }
```

Function returning by reference

- To provide usable access, **two** functions are required
  - Different headers, usually (syntactically) the same body
  - Global functions:

```
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]; }
```

  - Member functions:

```
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_;
};
```

Read-only arguments

- For read-only arguments passed by reference, **const is necessary**

  - Global function:

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

  - Member functions:

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

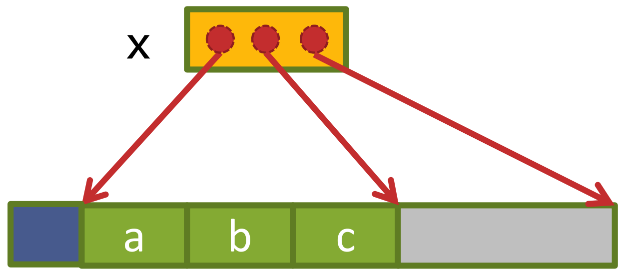
  - Otherwise, the argument could not be bound to an R-value

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

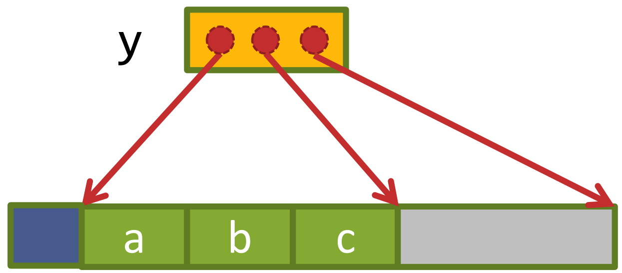
copy/move

Copy

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



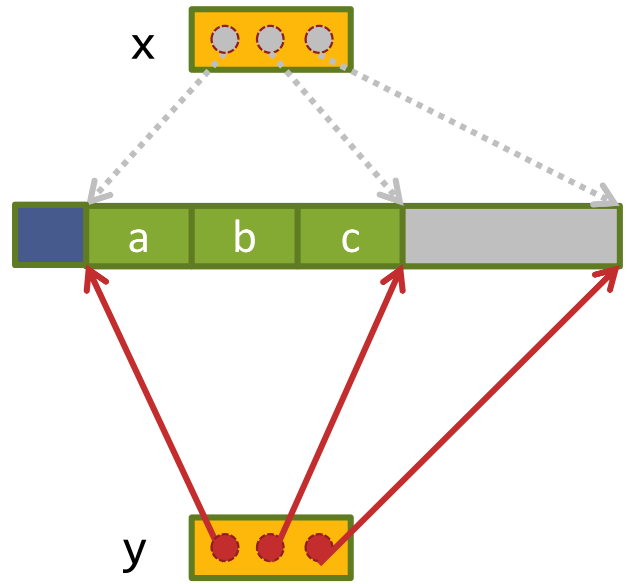

```
std::vector< char> y = x;
```



- A copy operation on containers and similar types
  - Requires allocation and copying of dynamically-allocated data
  - It is slow and may throw exceptions

Move

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




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

- After moving, the source is *empty*
  - Exact meaning depends on the type
- A move operation usually does no allocation
  - It is fast and does not throw exceptions

Move

- Move operation is invoked instead of copy, if
  - the source is explicitly marked with std::move(), **or**
  - the source is an **r-value**
    - temporary object, which cannot be accessed repeatedly
      - return values from functions which return by value
      - explicitly created temporary objects
      - results of casts etc.

Move

- The meaning of *copy* and *move* operations depends on the type
  - The behavior is implemented as four special member functions
    - copy-constructor – called when initializing a new object by copying

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

    - move-constructor – called when initializing a new object by moving

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

    - copy-assignment – called when copying a new value to an old object

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

    - move-assignment – called when moving a new value to an old object

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

  - if not implemented by the programmer, the compiler will create them
    - only if some (rather complex) conditions ensuring backward compatibility are met
      - otherwise the respective copy/move operations are not supported by the type
    - the compiler-generated implementation calls the corresponding functions for all data members (and base classes)
      - if you follow C++11 guidelines, this behavior will probably meet your needs
  - for elementary types (numbers, T *), *move* is implemented as *copy*
    - it may cause inconsistency between number and container members
  - when containers are *moved,* all elements are also moved
    - the source container becomes empty (except std::array which cannot be resized)

- 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() {}
```

Dynamic allocation

Dynamic allocation in C++11

- Use smart pointers instead of raw (T *) pointers

```
#include <memory>
```

  - one owner (pointer cannot be copied)
    - negligible runtime cost (almost the same as T *)

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

  - shared ownership
    - runtime cost of reference counting

```
void f() {
  std::shared_ptr< T> p = std::make_shared< T>();      // invokes new
  std::shared_ptr< T> q = p;         // pointer copied; object shared between q and p
}
```

- Memory is deallocated when the last owner disappears
  - Destructor of (or assignment to) the smart pointer invokes delete when required
  - Reference counting cannot deallocate cyclic structures

Using auto

- Compiler can infer the type of a variable from its initialization
  - 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
}
```

- Beware of conversions
  - k.size() returns std::size_t which may be larger than int (the type of 0)

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

Using pointers in modern C++

- Owner of object
  - std::unique_ptr< T>, std::shared_ptr< T>
  - Use only if objects must be allocated one-by-one
    - Possible reasons: Inheritance, irregular life range, graph-like structure, singleton
    - For holding multiple objects of the same type, use std::vector< T>
  - std::weak_ptr< T>
    - To enable circular references with std::shared_ptr< T>, used rarely
- Modifying observer
  - T *
    - In modern C++, native (raw, T*) pointers shall not represent ownership
  - Save T * in another object which needs to **modify** the T object
    - **Beware of lifetime**: The observer must stop observing before the owner dies
    - If you are not able to prevent premature owner death, you need shared ownership
- Read-only observer
  - const T *
  - Save const T * in another object which needs to **read** the T object
- Besides pointers, C++ has references (T &, const T &, T &&)
  - Used (by convention) for **temporary** access during a function call etc.

Owners and observers

- Example – unique ownership

```
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
```

- Owner pointers can point only to a complete dynamically allocated block
- Observer pointers can point to any piece of data anywhere
  - Parts of objects

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

  - Static data

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

  - Local data (beware: their lifetime is limited – avoid propagating observers outside of their scope)

```
void g( T * p);     // note: using reference (T &) instead of pointer is preferred here
void f() { T local_data; g( & local_data); }
```

- Dynamic allocation is slow
  - compared to static/automatic storage
  - the reason is cache behavior, not only the allocation itself
- Use dynamic allocation only when necessary
  - variable-sized or large arrays
  - polymorphic containers (containing various objects using inheritance)
  - object lifetimes not corresponding to function invocations
- Avoid data structures with individually allocated items
  - linked lists, binary trees, ...
    - std::list, std::map, ...
  - prefer contiguous structures (vectors, hash tables, B-trees, etc.)
  - avoiding is difficult - do it only if speed is important
- This is how C++ programs may be made faster than C#/java
  - C#/java requires dynamic allocation of every class instance

Pointer/reference conventions

Pointer/references

- C++ allows several ways of passing links to objects
  - smart pointers
  - C-like pointers
  - references
- Technically, all the forms allow almost everything
  - At least using dirty tricks to bypass language rules
  - Pointers require different syntax wrt. references
- By convention, the use of a specific form signalizes some intent
  - Conventions (and language rules) limits the way how the object is used
  - Conventions help to avoid "what-if" questions
    - What if someone destroys the object I am dealing with?
    - What if someone modifies the contents of the object unexpectedly?
    - ...

Passing a pointer/reference in C++ - conventions

	What the recipient may do?	For how long?	What the others will do meanwhile?
`std::unique_ptr<T>`	Modify the contents and destroy the object	As required	Nothing (usually)
`std::shared_ptr<T>`	Modify the contents	As required	Read/modify the contents
`T *`	Modify the contents	Until notified to stop/by agreement	Read/modify the contents
`const T *`	Read the contents	Until notified to stop/by agreement	Modify the contents
`T &`	Modify the contents	During a call/statement	Nothing (usually)
`T &&`	Steal the contents		Nothing
`const T &`	Read the contents	During a call/statement	Nothing (usually)

Multiple values in contiguous memory

Arrays and tuples

	Homogeneous (arrays)	Polymorphic (tuples)

Fixed size	`// modern: container-style`
	`static const std::size_t n = 3;`
	`std::array< T, n> a;`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`// structure/class`
		`struct S { T1 x; T2 y; T3 z; };`
		`S a;`
		
		`a.x = /*...*/;`
		`a.y.f();`

	`// native arrays (avoid!)`
	`static const std::size_t n = 3;`
	`T a[ n];`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`// for generic access`
		`std::tuple< T1, T2, T3> a;`
		
		`std::get< 0>( a) = /*...*/;`
		`std::get< 1>( a).f();`

Variable size	`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();`

```
std::array< T, 3>
```

or

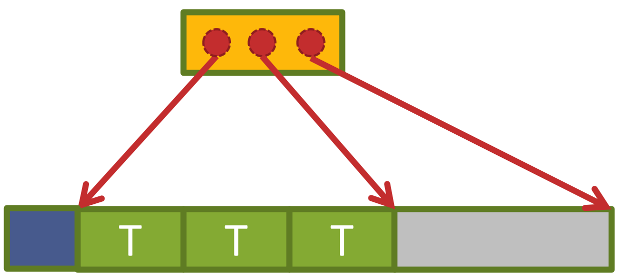
```
T[ 3]
```


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

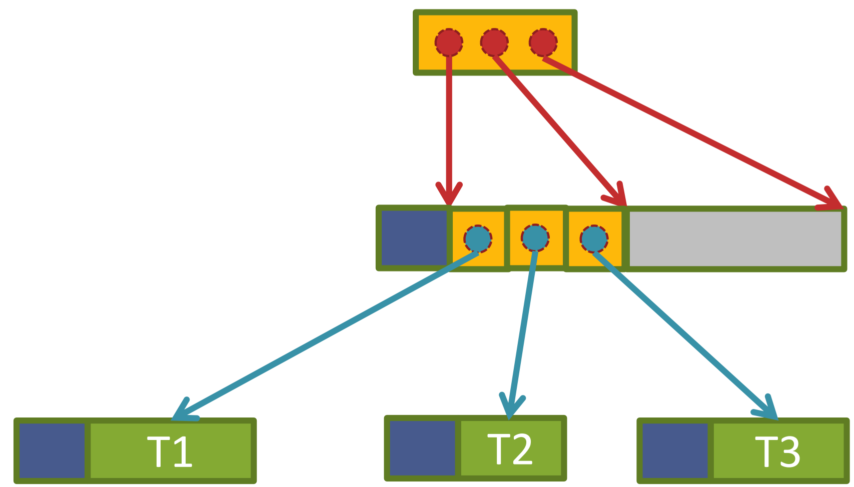
or

```
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="5">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="5">.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>

Smart Pointers - Examples

Transferring unique ownership

```
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);

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

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

Transferring unique ownership

```
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_;
};
```

Shared ownership

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

class recipient {
public:
  recipient( std::shared_ptr< channel> ch)
    : ch_( 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>( ch);
}

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

- The server and the recipient may be destroyed in any order
  - The last one will destroy the channel

Accessing without ownership transfer

```
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();
}
```

- The server and the recipient must be destroyed before the destruction of the channel

Holding pointers to locally allocated objects

```
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);
}
```

- The need to use "&" in constructor parameters warns of long life of the reference
  - "&" - converts reference to pointer
  - "*" - converts pointer to reference
- Local variables are automatically destructed in the reverse order of construction

Class holding a reference

```
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 and r will hold the reference to ch for their lifetime
  - There is no warning of that!
- If references are held by locally allocated objects, everything is OK
  - Destruction occurs in reverse order

ERROR: Passing a reference to local object out of its scope

```
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 will die sooner than s and r
  - s and r will access invalid object
  - Fatal crash sooner or later
- Nothing warns of this behavior
  - Prefer pointers in this case

ERROR: Killing an object in use

```
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 is destructed before s and r
  - Fatal crash sooner or later
- Rare programming practice

STL

Standard Template Library

STL

- Containers
  - Generic data structures
    - Based on arrays, linked lists, trees, or hash-tables
  - Store objects of given type (template parameter)

  - The container takes care of allocation/deallocation of the stored objects
    - All objects must be of the same type (defined by the template parameter)
      - Containers can not directly store polymorphic objects with inheritance
    - New objects are inserted by copying/moving/constructing in place
      - Containers can not hold objects created outside them

  - Inserting/removing objects: Member functions of the container
  - Reading/modifying objects: Iterators

STL – Example

```
#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
```

STL

- Sequential containers
  - New objects are inserted in specified location

  - array< T, N> - fixed-size array (no insertion/removal)
  - vector< T> - array, fast insertion/removal at the back end
    - stack< T> - insertion/removal only at the top (back end)
    - priority_queue< T> - priority queue (heap implemented in vector)
  - basic_string< T> - vektor s terminátorem
    - string = basic_string< char>
    - wstring = basic_string< wchar_t>
  - deque< T> - fast insertion/removal at both ends
    - queue< T> - FIFO (insert to back, remove from front)
  - forward_list< T> - linked list
  - list< T> - doubly-linked list

- Associative containers
  - New objects are inserted at a position defined by their properties
    - sets: type T must define ordering relation or hash function
    - maps: stored objects are of type pair< const K, T>
      - type K must define ordering or hash
    - multi-: multiple objects with the same (equivalent) key value may be inserted
  - Ordered (implemented usually by red-black trees)
    - set<T>
    - multiset<T>
    - map<K,T>
    - multimap<K,T>
  - Hashed
    - unordered_set<T>
    - unordered_multiset<T>
    - unordered_map<K,T>
    - unordered_multimap<K,T>

STL - Ordered Containers

- Ordered containers require ordering relation on the key type
  - Only < is used (no need to define >, <=, >=, ==, !=)
  - In simplest cases, the type has a built-in ordering

```
std::map< std::string, my_value> my_map;
```

  - If not built-in, ordering may be defined using a global function

```
bool operator<( const my_key & a, const my_key & b) { return /*...*/; }
std::map< my_key, my_value> mapa;
```

  - If global definition is not appropriate, ordering may be defined using a functor

```
struct my_functor {
  bool operator()( const my_key & a, const my_key & b) const { return /*...*/; }
};
std::map< my_key, my_value, my_functor> my_map;
```

  - If the ordering has run-time parameters, the functor will carry them

```
struct my_functor { my_functor( bool a); /*...*/ bool ascending; };
std::map< my_key, my_value, my_functor> my_map( my_functor( true));
```

STL - Unordered containers

- Hashed containers require two functors: hash function and equality comparison

```
struct my_hash {
  std::size_t operator()( const my_key & a) const { /*...*/ }
};
struct my_equal { public:
  bool operator()( const my_key & a, const my_key & b) const { /*return a == b;*/ }
};
std::unordered_map< my_key, my_value, my_hash, my_equal> my_map;
```

- If not explicitly defined by container template parameters, hashed containers try to use generic functors defined in the library
  - std::hash< K>
  - std::equal_to< K>
  - Defined for numeric types, strings, and some other library types

```
std::unordered_map< std::string, my_value> my_map;
```

STL – Iterators

- 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 – Iterators

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

- Every container defines functions to access both ends of the container
  - begin(), cbegin() - the first object (same as end() if the container is empty)
  - end(), cend() - the imaginary position behind the last object

```
  auto i1 = begin( c1);                          // also c1.begin()
  auto i2 = cbegin( c1);        // also c1.cbegin(), begin( c1), c1.begin()
  auto i3 = cbegin( c2);        // also c2.cbegin(), begin( c2), c2.begin()
```

- Associative containers allow searching
  - find(k) - first object equal (i.e. not less and not greater) to k, end() if not found
  - lower_bound(k) - first object not less than k , end() if not found
  - upper_bound(k) - first object greater than k , end() if not found

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

- Iterators may be shifted to neighbors in the container
  - all iterators allow shifting to the right and equality comparison

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

  - bidirectional iterators (all containers except forward_list) allow shifting to the left

```
  -- i1;
```

  - random access iterators (vector, string, deque) allow addition/subtraction of integers, difference and comparison

```
  auto delta = i4 - c1.begin();  // number of objects left to i4; my_container::difference_type === std::ptrdiff_t
  auto i7 = c1.end() - delta;    // the same distance from the opposite end; my_container::iterator
  if ( i4 < i7 )
    auto v = i4[ delta].second;  // same as (*(i4 + delta)).second, (i4 + delta)->second
}
```

STL – Iterators

- Caution:
  - Shifting an iterator before begin() or after end() is illegal

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

  - Comparing iterators associated to different (instances of) containers is illegal

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

  - Insertion/removal of objects in vector/basic_string/deque **invalidate** all associated iterators
    - The only valid iterator is the one returned from insert/erase

```
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
```

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;
  }
}
```

Algorithms

Algorithms

- Set of generic functions working on containers
  - cca 90 functions, trivial or sophisticated (sort, make_heap, set_intersection, ...)

```
#include <algorithm>
```

  - Containers are accessed indirectly - using iterators
    - Typically a pair of iterator specifies a range inside a container
    - Algorithms may be run on complete containers or parts
    - Anything that looks like an iterator may be used
  - Some algorithms are read-only
    - The result is often an iterator
    - E.g., searching in non-associative containers
  - Most algorithms modify the contents of a container
    - Copying, moving (using std::move), or swapping (using std::swap) elements
    - Applying user-defined action on elements (defined by functors)
  - Iterators does not allow insertion/deletion of container elements
    - The space for "new" elements must be created before calling an algorithm
    - Removal of unnecessary elements must be done after returning from an algorithm

Algorithms

- Iterators does not allow insertion/deletion of container elements
  - The space for "new" elements must be created before calling an algorithm

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

  - Note: This example does not require algorithms:

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

  - Removal of unnecessary elements must be done after returning from an algorithm

```
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 – Algorithms

- Fake iterators
  - Algorithms may accept anything that works like an iterator
  - The required functionality is specified by iterator category
    - Input, Output, Forward, Bidirectional, RandomAccess
  - Every iterator must specify its category and some other properties
    - std::iterator_traits
    - Some algorithms change their implementation based on the category (std::distance)

  - Inserters

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

  - Text input/output

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

C++20 - ranges

- [C++20] – a pair of iterators replaced by a ***range***
  - *range* is anything equipped with begin() and end()
    - Any container is a *range* – this kind of range is the owner of the data!
      - copying such a range copies the data
    - *view* range is a reference to the data – not the owner
      - view range may be copied in constant time
      - all_view(k) is a reference to all elements in a container
      - iota_view(10,20) is a virtual container containing [10,11,...,19]
  - *range adaptor* allows filtration or transformation of data
    - *filter_view(range, pred)* returns only the elements of *range* which satisfy *pred*
    - adapters may be also applied using unix-like pipe syntax:

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

  - Existing algorithms will be presented also with *range* interfaces
  - *Range* fits into the [C++11] range-based for
  - *There is no complete implementation of ranges yet (November 2019)*
    - *ranges* require *concepts* which themselves are a major language extension [C++20]

Functors

STL – Functors

- Example - 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 may be anything that has the function call operator f(x)
  - a global function (pointer to function), or
  - a *functor*, i.e. a class containing operator()
- The function f (its operator()) is called for each element in the given range
  - The element is accessed using the * operator which typically return a reference
  - The function f can modify the elements of the container

STL – Algorithms

- A simple application of 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
  - New syntax construct - generates a functor

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

STL – Algorithms

- Passing parameters requires a functor

```
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));
}
```

- Equivalent implementation using lambda

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

STL – Algoritmy

- A functor may modify its contents

```
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;
}
```

- Using lambda (the generated functor contains a reference to s)

```
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 expression

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

  - Declares a class similar to this sketch:

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

    - vlist determined by local variables used in the *body*
    - TList determined by their types and adjusted by *capture*
    - operator() is const if *mutable* not present

  - The lambda expression corresponds to creation of an anonymous object

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

C++11

Lambda expressions – return types

C++11

- Return type of the operator()
  - Explicitly defined

```
[]() -> int { /*…*/ }
```

  - Automatically derived if body contains just one return statement

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

  - void otherwise

Lambda expressions – capture

C++11

- Capture
  - Defines which external variables are accessible and how
    - local variables in the enclosing function
    - *this*, if used in a member function
  - Determines the data members of the functor
  - Explicit capture
    - The external variables explicitly listed in *capture*

```
[a,&b,c,&d,this]
```

    - variables marked & passed by reference, the others by value
    - when returning lambdas from functions, beware of the lifetime of referenced variables
  - Implicit capture
    - The required external variables determined automatically by the compiler, *capture* defines the mode of passing

```
[=]
[=,&b,&d]
```

    - passed by value, the listed exceptions by reference

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

    - passed by reference, the listed exceptions by value

Class

Class

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

- Class in C++ is an extremely powerful construct
  - Other languages often have several less powerful constructs (class+interface)
  - Requires caution and conventions
- Three degrees of usage
  - Non-instantiated class - a pack of declarations (used in generic programming)
  - Class with data members
  - Class with inheritance and virtual functions (object-oriented programming)
- class = struct
  - struct members are by default public
    - by convention used for simple or non-instantiated classes
  - class members are by default private
    - by convention used for large classes and OOP

Three degrees of classes

Non-instantiated class

```
class X {
public:
  typedef int t;
  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() {}
  void f()
  { f_(); }
private:
  virtual void f_() = 0;
};

class V : public U {
public:
  V() : m_( 0) {}
private:
  int m_;
  virtual void f_()
  { ++ m_; }
};
```

Type and static members of classes

```
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);
}
```

- Type and static members...
  - Nested class definitions
  - typedef/using definitions
  - static member constants
  - static member functions
  - static member variables
- ... are not bound to any class instance (object)
- Equivalent to global types/variables/functions
  - But referenced using qualified names (prefix X::)
  - Encapsulation in a class avoids name clashes
    - But namespaces do it better
  - Some members may be private
  - Class may be passed to a template

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 constexpr t c = 1;
  static t f( N p);
private:
  static t v;     // declaration of X::v
};
```

- Class must be defined in one piece
  - Except of definitions placed outside

```
X::t X::v = X::c;                           // definition of X::v
X::t X::f( N p) { return p.m + v; }         // definition of X::f
```

- Access to members requires qualified names

```
void f2()
{
  X::N a;
  auto b = X::f( a);
}
```

- A class may become a template argument
  - This is the (only) reason for uninstantiated classes

```
using my_class = some_generic_class< X>;
```

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;
  constexpr t c = 1;
  extern t v;           // declaration of X::v
};
```

- Namespace may be reopened and member declarations added
  - Namespace may be split into several header files

```
namespace X {
  inline t f( N p) { return p.m + v; }
};
```

  - Definitions of previously declared namespace members may be outside

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

Namespaces

- 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;
  constexpr t c = 1;
  extern t v;          // declaration of X::v
};
```

- Namespace may be reopened and member declarations added
  - Namespace may be split into several header files

```
namespace X {
  inline t f( N p) { return p.m + v; }
};
```

  - Definitions of previously declared namespace members may be outside

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

```
void f2()
{
  X::N a;
```

  - Functions in namespaces are visible by *argument-dependent lookup*

```
  auto b = f( a);
```

    - calls X::f because the class type of a is a member of X

  - Namespace members can be made directly visible

```
  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 (smart) pointer

```
auto p = std::make_unique< Y>();
auto 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;
auto 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(); // read-only observer
pp->set_m(0)        // error
```

Inheritance and virtual functions

Inheritance

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

- Derived class is a descendant of Base class
  - Contains all types, data elements and functions of Base
    - Because of this, a pointer/reference to Derived may be silently converted to a pointer/reference to Base
    - The opposite conversion is available as explicit cast
  - New types/data/functions may be added
    - Hiding old names by new names is not wise, except for virtual functions
  - Functions declared as virtual in Base may change their behavior by reimplementation in Derived

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

Classes in inheritance

- Abstract class
  - Definition in C++: A class that contains some pure virtual functions

```
virtual void f() = 0;
```

    - Such class are incomplete and cannot be instantiated alone
  - General definition: A class that will not be instantiated alone (even if it could)
  - Defines the interface which will be implemented by the derived classes

- Concrete class
  - A class that will be instantiated as an object
  - Implements the interface required by its base class

Virtual functions

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

- Virtual function call works only in the presence of pointers or references

```
std::unique_ptr<Base> p = std::make_unique< Derived>();      // automatic conversion
p->f();   // calls Derived::f although p is pointer to Base
```

- Without pointers/references, having functions virtual has no sense

```
Derived d;
d.f();    // calls Derived::f even for non-virtual f

Base b = d;          // slicing = copying a part of an object
b.f();    // calls Base::f even for virtual f
```

- Slicing is specific to C++

Inheritance and the destructor

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

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

- Old-style

```
Base * p = new Derived;
delete p;
```

- Modern-style

```
{
  std::unique_ptr<Base> p =
    std::make_unique< Derived>();
  // destructor of unique_ptr calls delete
}
```

- Language rule:
  - If an object is destroyed using a pointer to a base class, the base class must have a *virtual* destructor
  - This triggers the more complex implementation of delete:
    - Correctly destruct the complete object
    - Correctly determine the memory block
- Recommendation:
  - Every abstract class shall have a virtual destructor
    - Cost is negligible because other virtual functions are present
    - A pointer to the abstract class will likely be used for destruction

Inheritance

- Inheritance mechanisms in C++ are very strong
  - Often misused

- Inheritance shall be used only in these cases

  - ISA hiearachy
    - Eagle IS A Bird
    - Square-Rectangle-Polygon-Drawable-Object

  - Interface-implementation
    - Readable-InputFile
    - Writable-OutputFile
    - (Readable+Writable)-IOFile

Inheritance

- ISA hierarchy
  - C++: Single non-virtual public inheritance

    ```
    class Derived : public Base
    ```

  - Abstract classes may contain data (although usually do not)

- Interface-implementation
  - C++: Multiple virtual public inheritance

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

    - virtual inheritance merges copies of a base class multiply included via diamond patterns
  - Abstract classes usually contain no data
  - Interfaces are (typically) not used to own (destroy) the object

- Often combined

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

Misuse of inheritance

- Misuse of inheritance - #1

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

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

- Leads to slicing:

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

Complex x;

double a = abs( x);          // it CAN be compiled - but it should not
```

- Reference to the derived class may be assigned to a reference to the base class
  - Complex => Complex & => Real & => const Real &

Misuse of inheritance

- Misuse of inheritance - #2

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

- Mistake: Objects in C++ are not mathematical objects

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

Real x;
set_to_i( x);      // it CAN be compiled - but it should not
```

- Real => Real & => Complex &

Two worlds of classes in C++

Classes without inheritance

- No virtual functions
- No visible pointers usually required
  - When multiple objects exist
    - Allocated usually via containers

```
std::vector< MyClass> k;
```

  - When standalone

```
MyClass c;
```

- If ownership must be transferred, moving may be used

```
std::vector< MyClass> k2 = move( k);
MyClass c2 = std::move( c);
```

- Move required
  - For insertion into containers
  - For transfer of ownership
- Copy often required too
- Individual allocation required only if
  - ownership must be transferred
  - and observers are required

```
auto p = std::make_unique< MyClass>();
MyClass * observer = p.get();
auto p2 = move( p);
```

Classes with inheritance

- Concrete classes of different size and layout
  - Usually mixed in a data structure
  - Cannot be allocated in a common block
  - Individual dynamic allocation
- Common base class
  - Serves as a unified handle for different concrete classes
  - Pointers required

```
std::vector< std::unique_ptr< Base>> k;
```

  - Virtual destructor required
- Copy/move not required/supported
  - Pointers are copied/moved instead
  - Objects often have identity

Special member functions

Constructors and destructors

- Constructor of class T is a method named T
  - Return type not specified
  - More than one constructor may exist with different arguments
  - Never virtual
  - A constructor is called whenever an object of the type T is created
    - Constructor parameters specified in the moment of creation
    - Some constructors have special meaning
    - Some constructors may be generated by the compiler
  - Constructors cannot be called directly
- Destructor of class T is a method named ~T
  - No arguments, no return value
  - May be virtual
  - The destructor is called whenever an object of the type T is destroyed
    - The destructors may be generated by the compiler
  - Explicit call must use special syntax 

Special member functions

- Default constructor

```
T();
```

  - For object without explicit initialization
  - Generated by compiler if required and if the class has no constructor at all:
    - Data members of non-class types are not initialized
    - Data members of class types and base classes are initialized by calling their default constructors
    - Generation may fail due to non-existence or inaccessibility of element constructors

- Destructor

```
~T();
```

  - Generated by compiler if required and not defined
    - Calls destructors of data members and base classes
  - If a class derived from T has to be destroyed using T *, the destructor of T must be virtual
    - All abstract classes shall have a virtual destructor

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

copy/move

- Special member functions
  - Copy constructor

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

  - Move constructor

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

  - Copy assignment operator

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

  - Move assignment operator

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

copy/move

- Compiler-generated implementation
  - Copy constructor

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

    - applies copy constructor to every element
  - Move constructor

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

    - applies move constructor to every element
  - Copy assignment operator

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

    - applies copy assignment to every element
  - Move assignment operator

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

    - applies move assignment to every element

  - elements are data members and base classes
  - for elements of non-class types, move is equivalent to copy

  - the default keyword allows to enforce generation by the compiler

copy/move

- If needed, compiler will generate the methods automatically under these conditions:
  - Copy constructor/assignment operator
    - if there is no definition for the method and no move method is defined
    - this is backward-compatibility rule; future development of the language will probably make the condition more stringent (no copy/move/destructor at all)
  - Move constructor/assignment operator
    - if no copy/move method is defined and no destructor is defined
- the default keyword overrides the conditions

Conversions

Special member functions

- Conversion constructors

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

  - Generalized copy constructor
  - Defines conversion from U to T
  - If conversion effect is not desired, all one-argument constructors must be "explicit":

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

- Conversion operators

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

  - Defines conversion from T to U
  - Returns U by value (using copy-constructor of U, if U is a class)

- Compiler will never use more than one user-defined conversion in a chain

Type cast

- Various syntax styles
  - C-style cast

```
(T)e
```

    - Inherited from C
  - Function-style cast

```
T(e)
```

    - Equivalent to (T)e
    - T must be single type identifier or single keyword
  - Type conversion operators
    - Differentiated by intent (strength and associated danger) of cast:

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

    - New - run-time assisted cast:

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

Dynamic cast

`dynamic_cast<T>(e)`

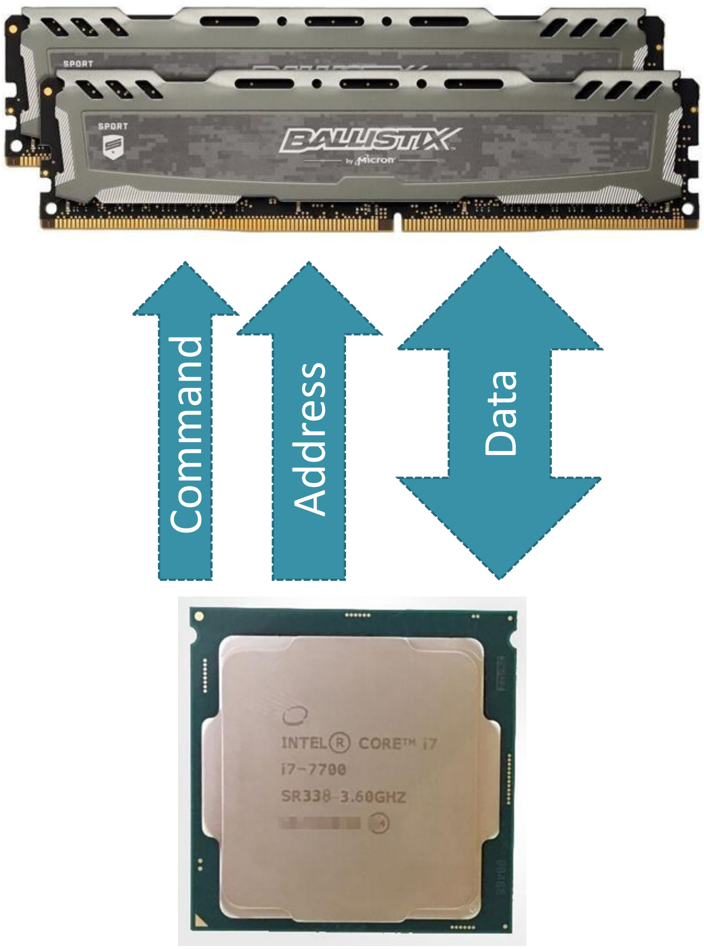
- Most frequent use
  - Converting a pointer to a base class to a pointer to a derived class

```
class Base { public:
  virtual ~Base(); /* base class must have at least one virtual function */
};
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 ) { /* ... */ }
```

Everything you had to know about the Principles of Computers

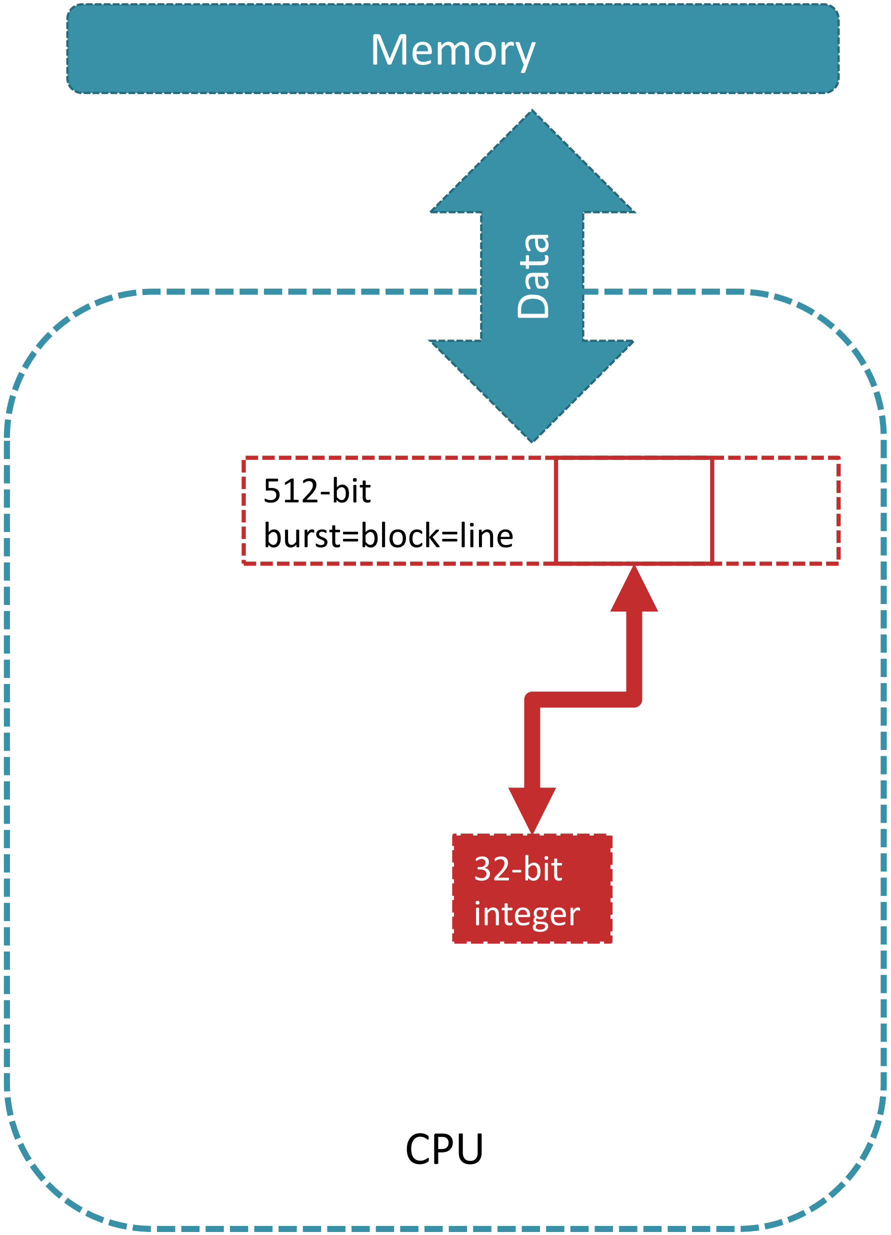
(but were afraid to ask)

Minimal computer = CPU + RAM



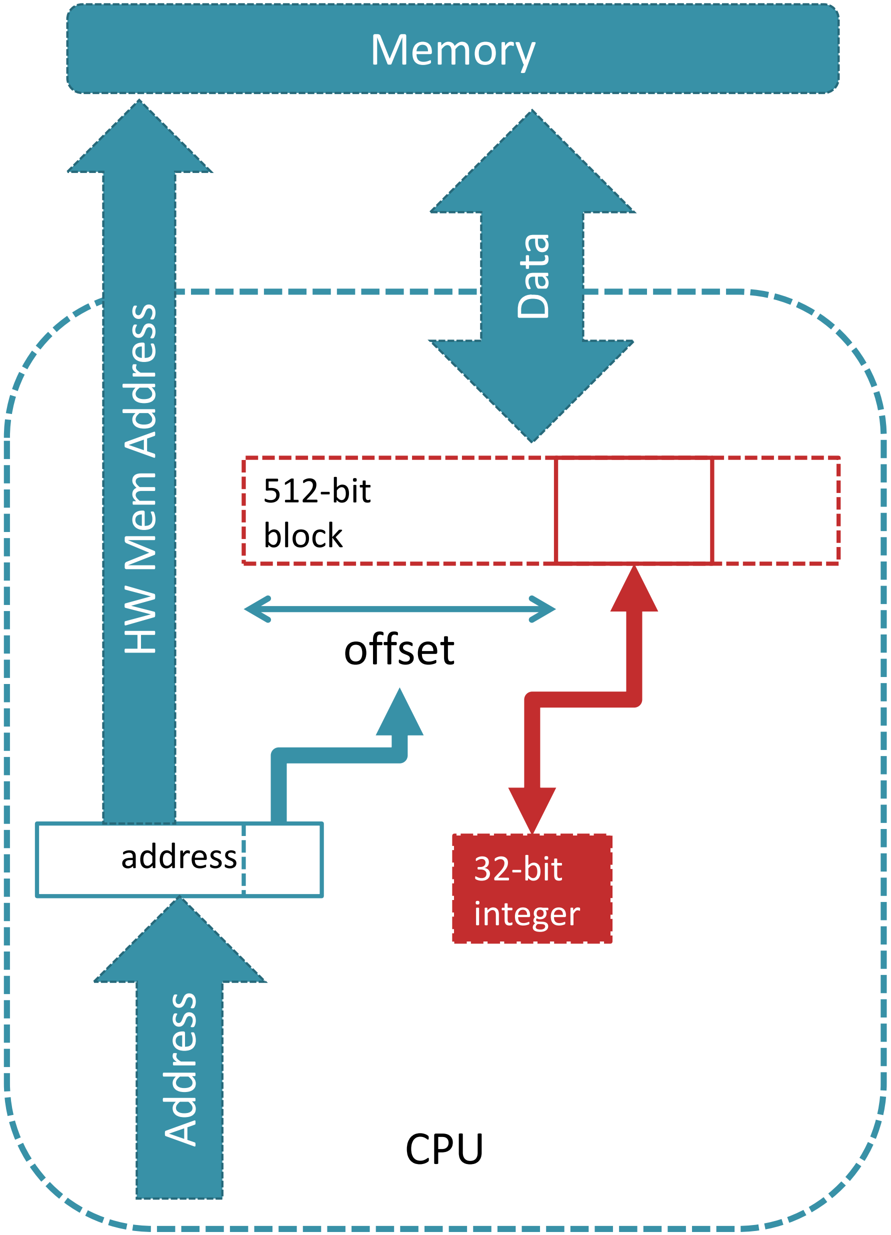
- CPU accesses memory using *commands*
  - Read/Write
  - Each command transfers a *burst* of data from/to memory
    - 512 bits in recent DDR4 chips
  - Address determines which 512-bit block is accessed
    - 16 GB = 256M * 512 bit
      - 28 address bits needed for 16 GB
    - Memory sizes are marketed 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



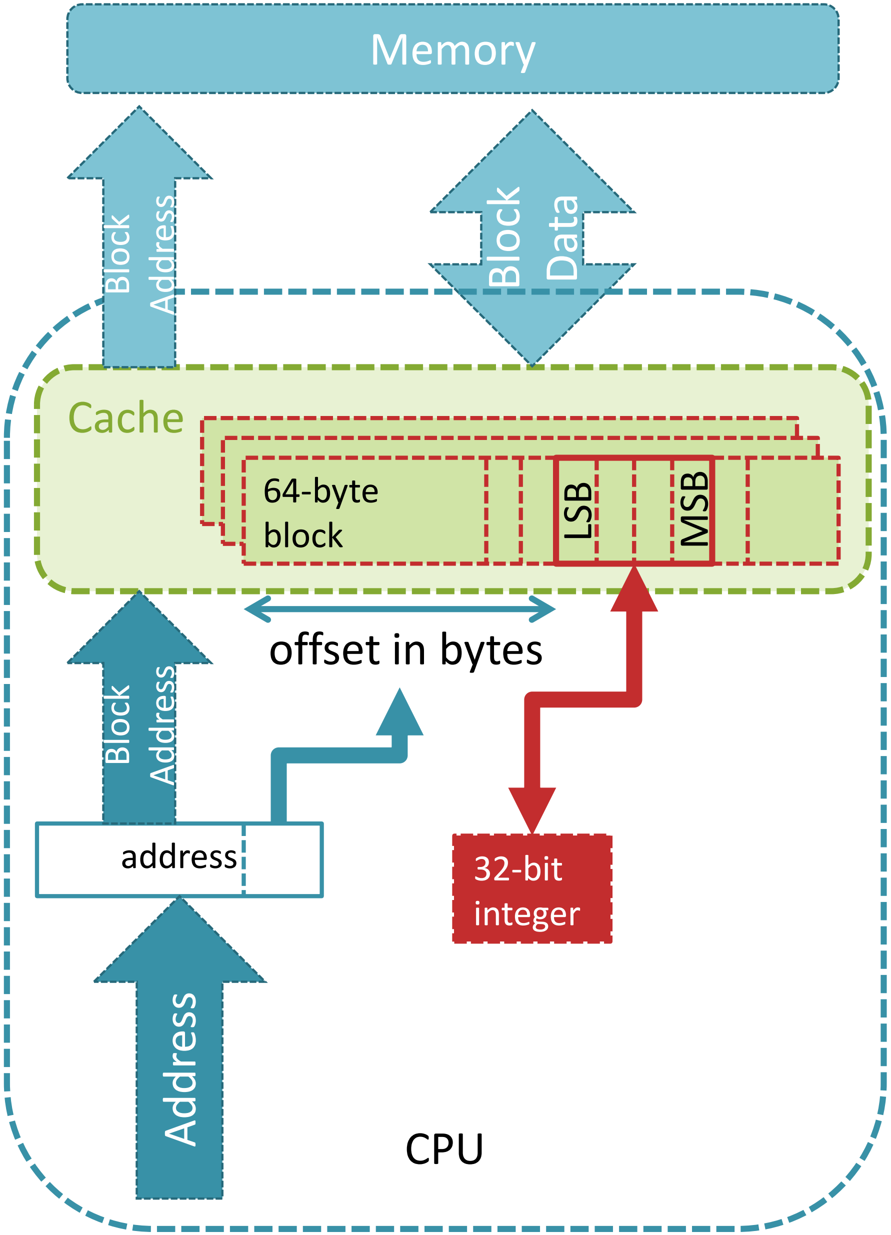
- CPU can work with *elementary data types*
  - Integers of various widths
    - today: 8, 16, 32, 64 bits
  - Floating-point in different formats
    - Intel/AMD: 32, 64, 80 bits
- CPU can read/write the elementary data types
  - only at some positions wrt. the 512-bit memory blocks
  - any elementary 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



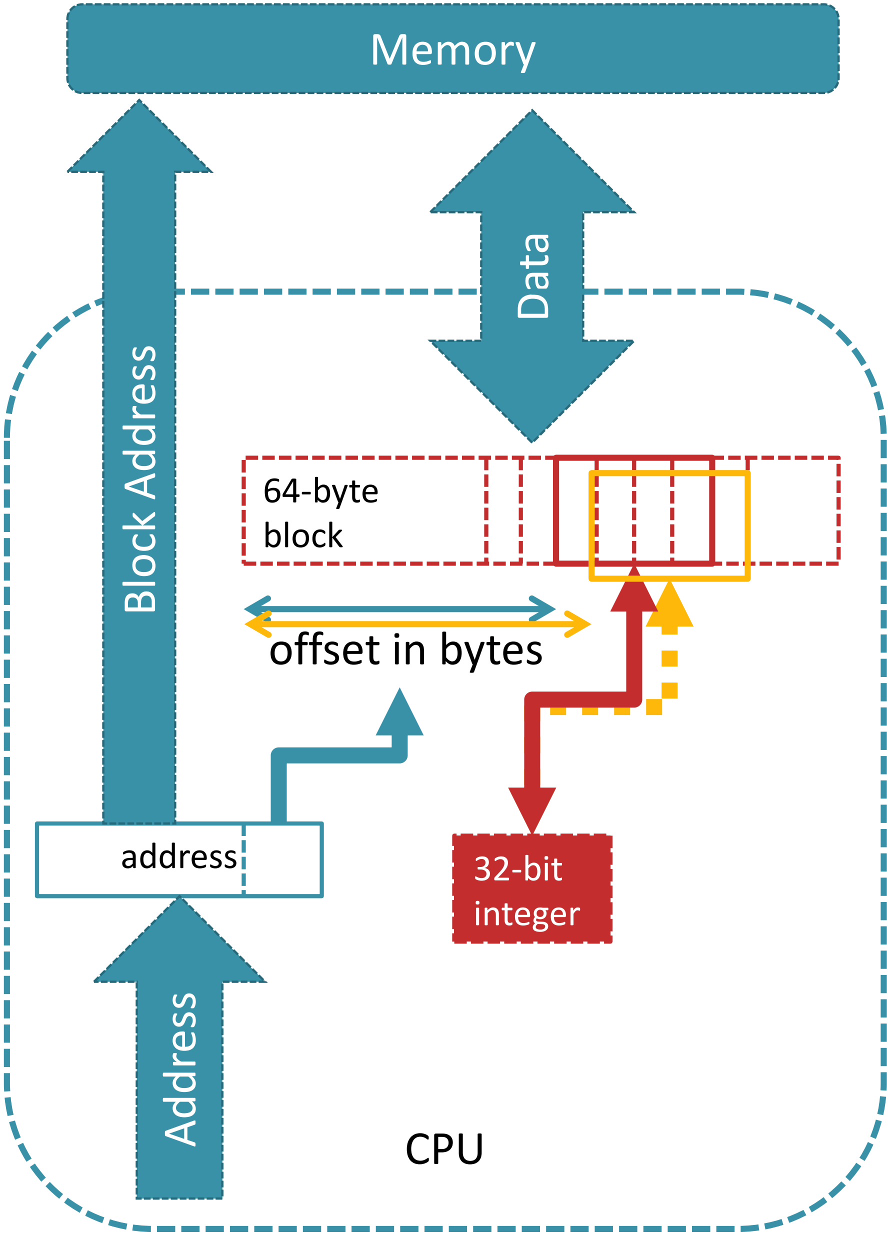
- CPU can internally read/write the elementary data types
  - the offset inside block is determined by some lower bits of the address
- Address granularity
  - how many bits are skipped when address is incremented by 1
  - 1 bit granularity is impractical
  - early computers used the size of their floating-point 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



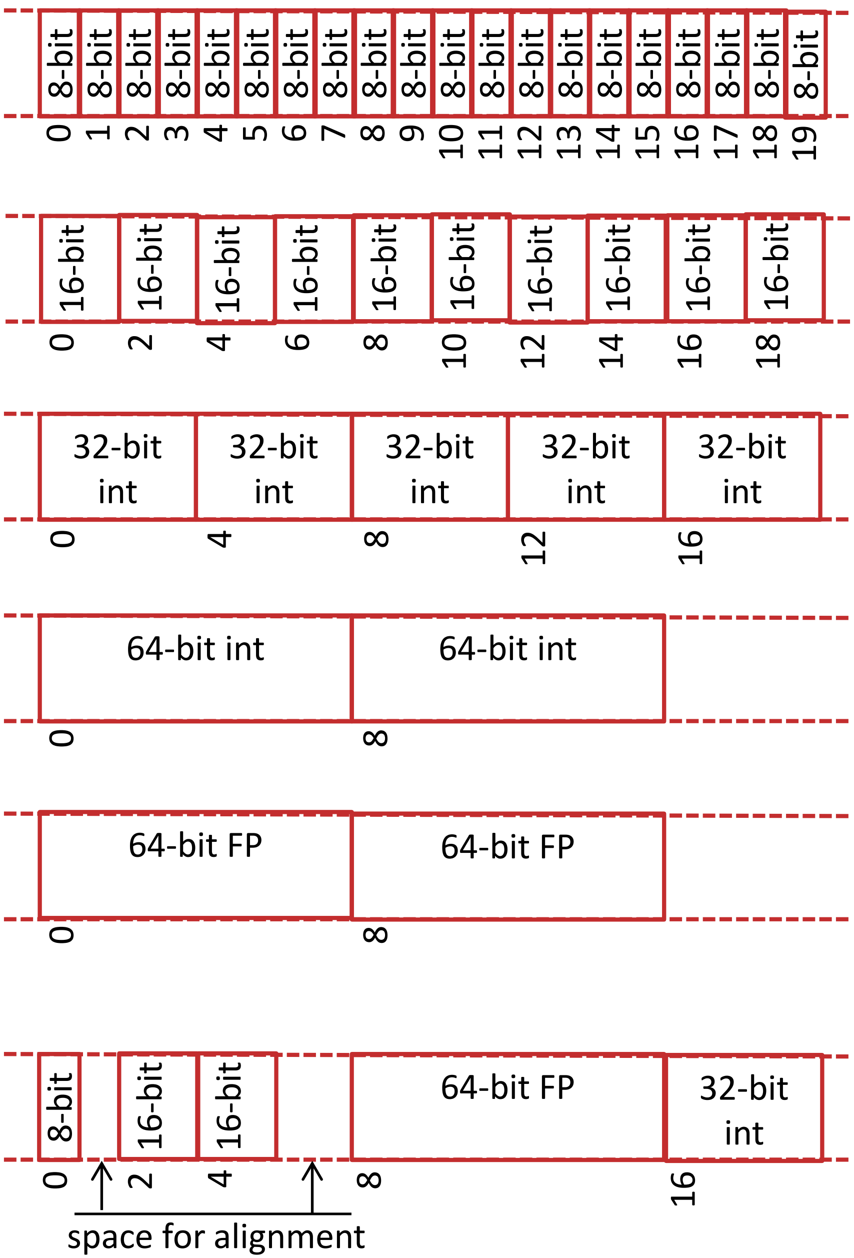
- 8 bit address granularity
  - no way to directly address bits
    - difficult implementation of bit arrays
    - no pointers/references to bits
  - most elementary data types span several bytes – order must be defined
    - Intel/AMD: *little-endian* – lower bits at lower addresses
    - software can 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



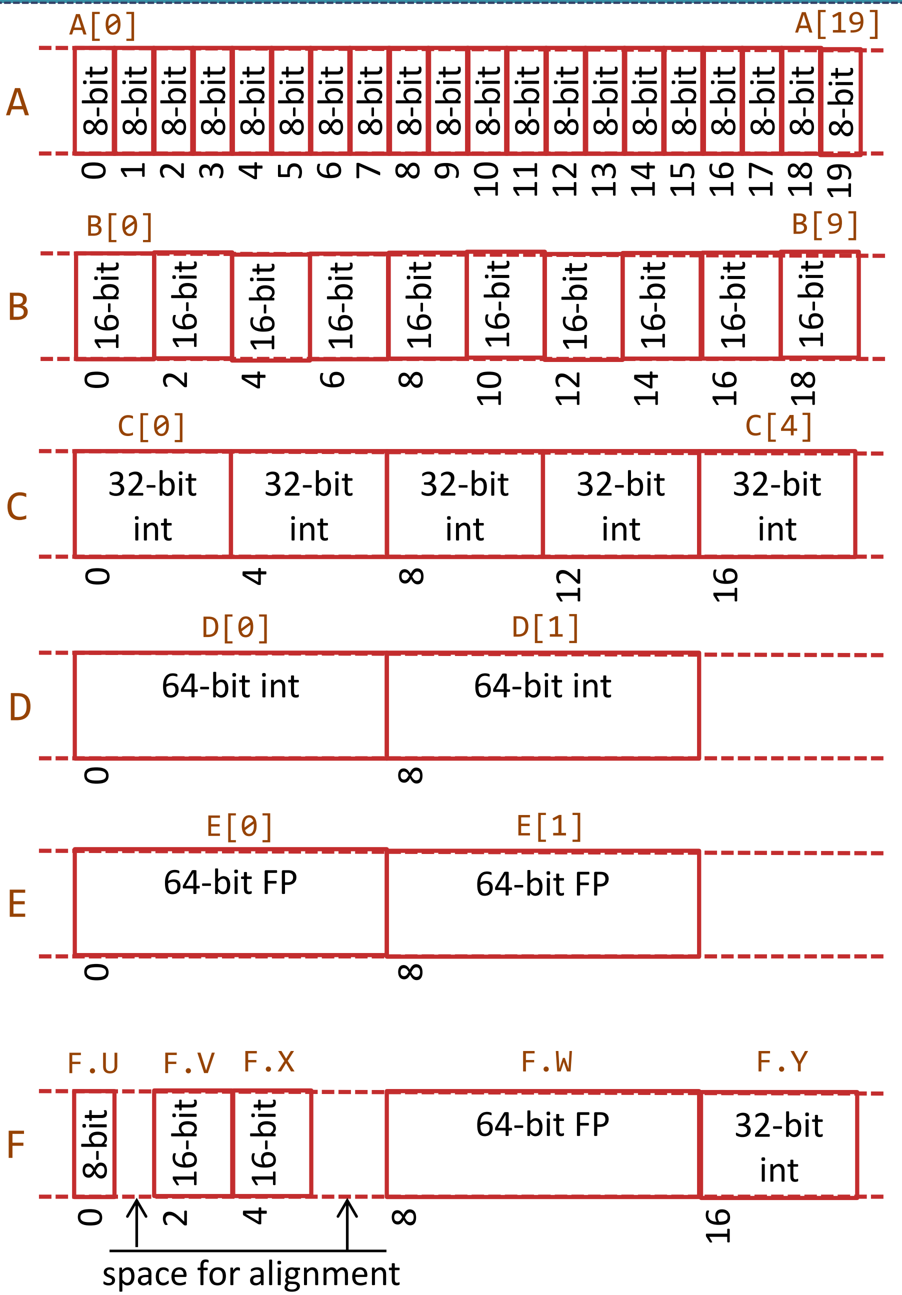
- CPU can read/write the elementary data types
  - only at some addresses wrt. the 64-byte memory blocks
  - *aligned* addresses: offset is a multiple of data type size
    - CPU contains hardware for quick access to these positions
  - *unaligned* addresses
    - Intel/AMD: access is possible at any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

Data in memory – arrays and structures



- Complex (*aggregate*) data
  - Illusion created by compilers
    - No hardware support at all
  - Elementary data grouped together
    - Placed at adjacent addresses
      - With alignment gaps
    - Locality improves cache-hit ratio
  - Two ways of grouping
    - *Arrays* – homogeneous
      - Allows run-time indexing
    - *Structures* – heterogeneous
  - All higher-language concepts are based on arrays and structures
    - With the addition of pointers
    - Class = structure + type info
      - In C++, type info is optional

Data in memory – arrays and structures in C++



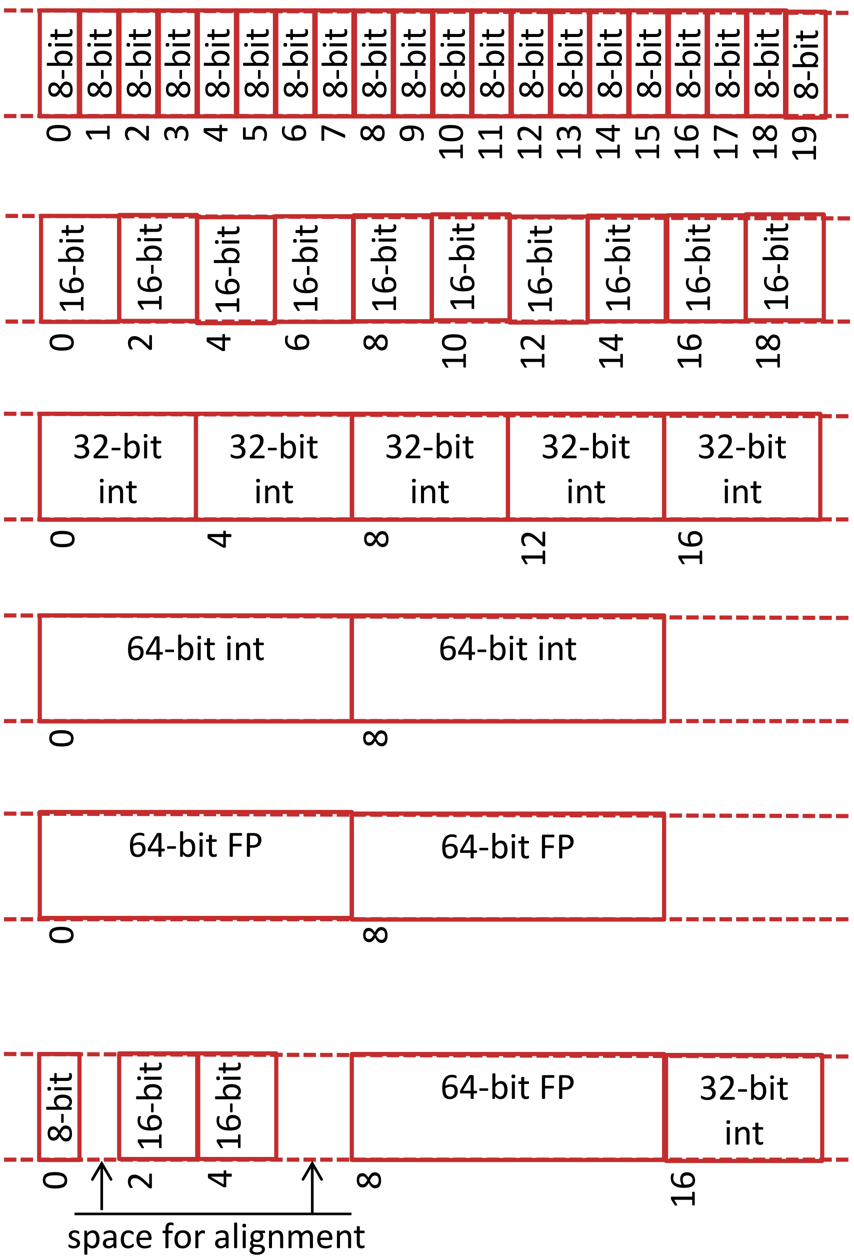
- Arrays
  - N elements of the same type

```
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;
```

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

Signed integer types – names in C/C++

<table>
<tr><th>Signed integer types</th><th rowspan="2">Not guaranteed to exist (on exotic architectures)</th><th rowspan="2">Not guaranteed to exactly match (on exotic architectures)</th><th>x86 (32-bit)</th><th colspan="2">x64 (64-bit)</th></tr>
<tr><th>Number of bits</th><th colspan="2">MS Visual C++</th><th>GNU C++</th></tr>
<tr><td>8</td><td>std::int8_t</td><td>std::int_least8_t</td><td colspan="3">signed char
std::int_fast8_t</td></tr>
<tr><td>16</td><td>std::int16_t</td><td>std::int_least16_t</td><td colspan="3">[signed] short [int]</td></tr>
<tr><td rowspan="3">32</td><td rowspan="3">std::int32_t</td><td rowspan="3">std::int_least32_t</td><td colspan="3">[signed] int</td></tr>
<tr><td colspan="2">[signed] long [int]
std::int_fast16_t
std::int_fast32_t</td><td></td></tr>
<tr><td>std::ptrdiff_t</td><td colspan="2"></td></tr>
<tr><td rowspan="3">64</td><td rowspan="3">std::int64_t</td><td rowspan="3">std::int_least64_t</td><td colspan="3">[signed] long long [int]
std::int_fast64_t</td></tr>
<tr><td colspan="2"></td><td>[signed] long [int]
std::int_fast16_t
std::int_fast32_t</td></tr>
<tr><td></td><td colspan="2">std::ptrdiff_t</td></tr>
</table>

- 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++

<table>
<tr><th>Signed integer types</th><th rowspan="2">Not guaranteed to exist (on exotic architectures)</th><th rowspan="2">Not guaranteed to exactly match (on exotic architectures)</th><th>x86 (32-bit)</th><th colspan="2">x64 (64-bit)</th></tr>
<tr><th>Number of bits</th><th colspan="2">MS Visual C++</th><th>GNU C++</th></tr>
<tr><td>8</td><td>std::int8_t</td><td>std::int_least8_t</td><td colspan="3">signed char
std::int_fast8_t</td></tr>
<tr><td>16</td><td>std::int16_t</td><td>std::int_least16_t</td><td colspan="3">[signed] short [int]</td></tr>
<tr><td rowspan="3">32</td><td rowspan="3">std::int32_t</td><td rowspan="3">std::int_least32_t</td><td colspan="3">[signed] int</td></tr>
<tr><td colspan="2">[signed] long [int]
std::int_fast16_t
std::int_fast32_t</td><td></td></tr>
<tr><td>std::ptrdiff_t</td><td colspan="2"></td></tr>
<tr><td rowspan="3">64</td><td rowspan="3">std::int64_t</td><td rowspan="3">std::int_least64_t</td><td colspan="3">[signed] long long [int]
std::int_fast64_t</td></tr>
<tr><td colspan="2"></td><td>[signed] long [int]
std::int_fast16_t
std::int_fast32_t</td></tr>
<tr><td></td><td colspan="2">std::ptrdiff_t</td></tr>
</table>

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

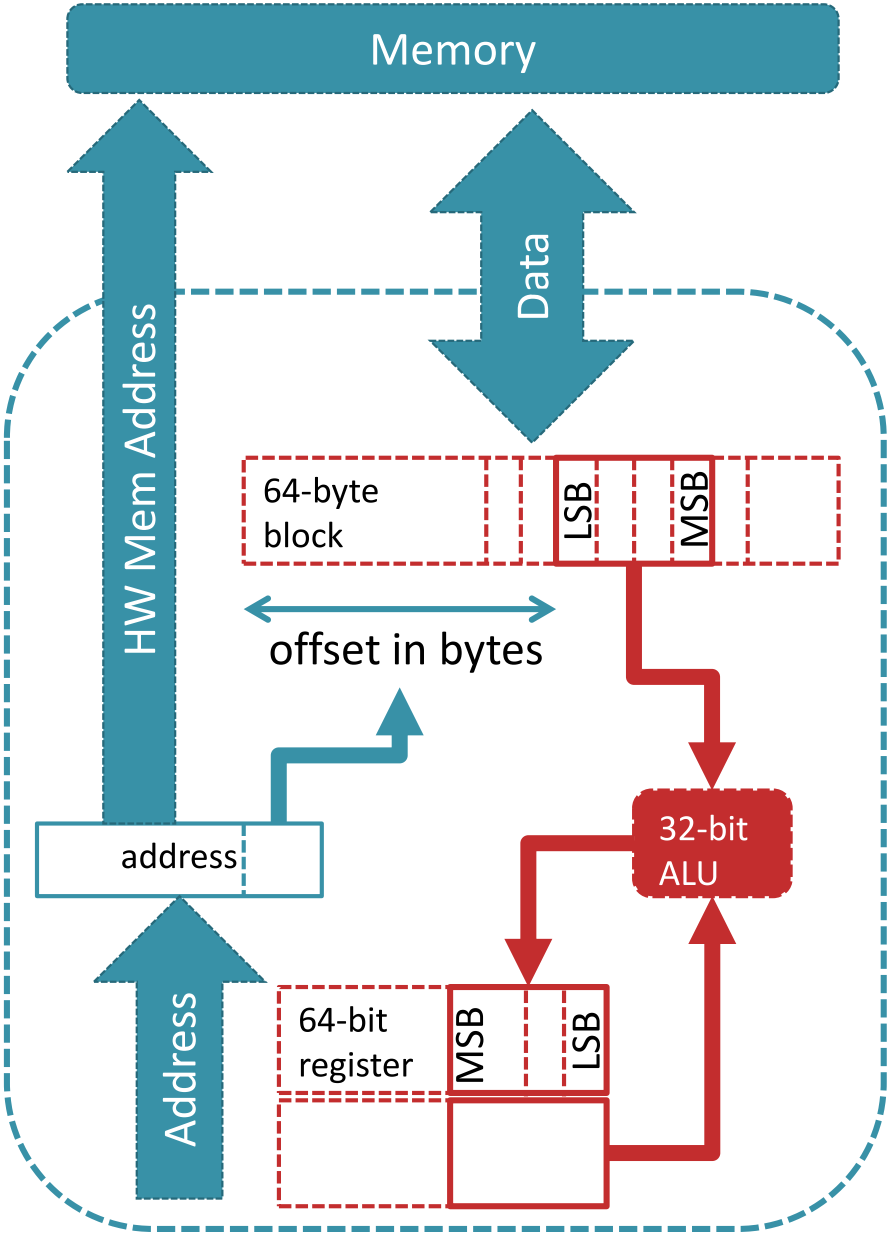
```
class C { public:
  std::string_view get_s() {
    return m_s;
  }
  void set_s( std::string_view p) {
    m_s = p;
  }
private:
  std::string m_s = “Hello”;
};
void f() {
  C obj;
  std::string_view s = obj.get_s();
  std::cout << s;         // OK
  obj.set_s( “Welcome”);
  std::cout << s;     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

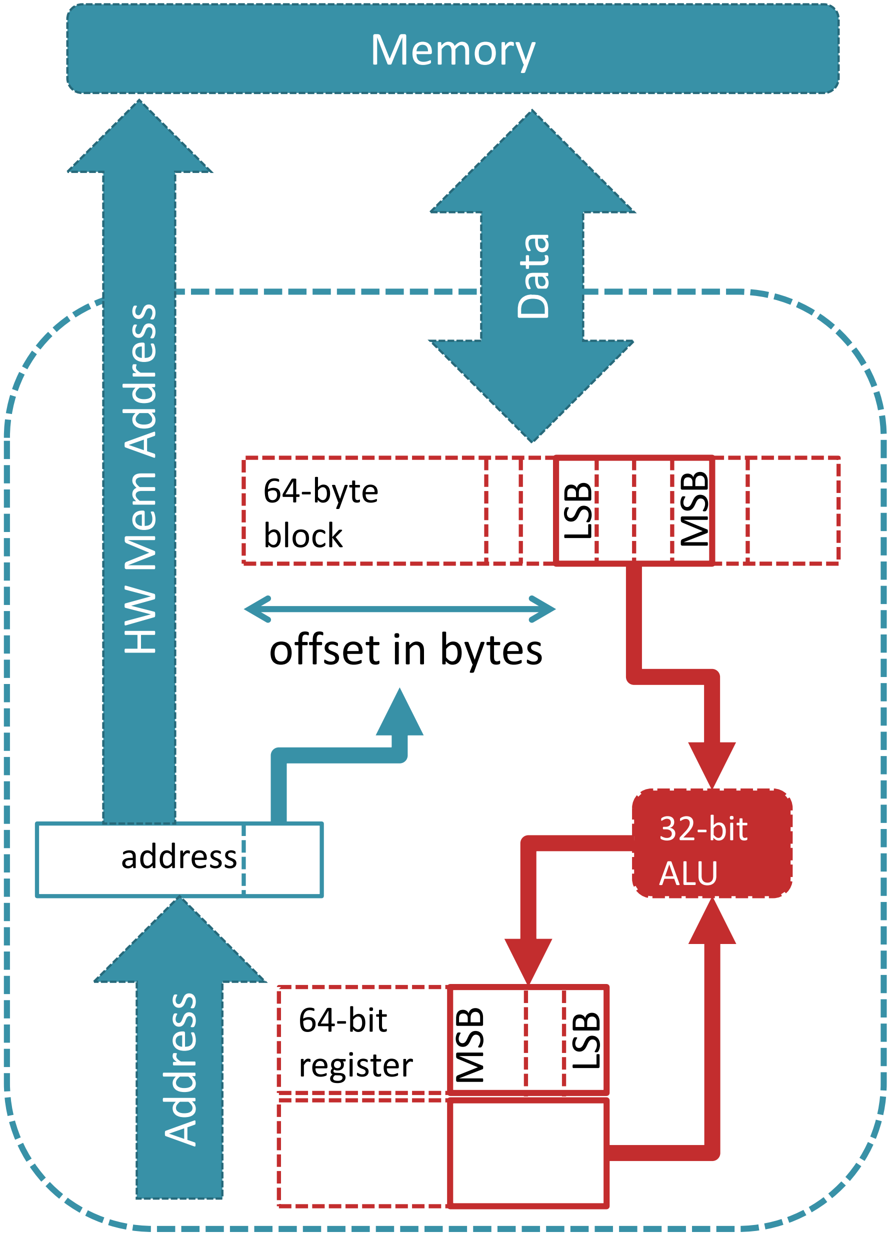
Everything you had to know about the Principles of Computers

(part 2 - addresses)

Inside CPU – registers

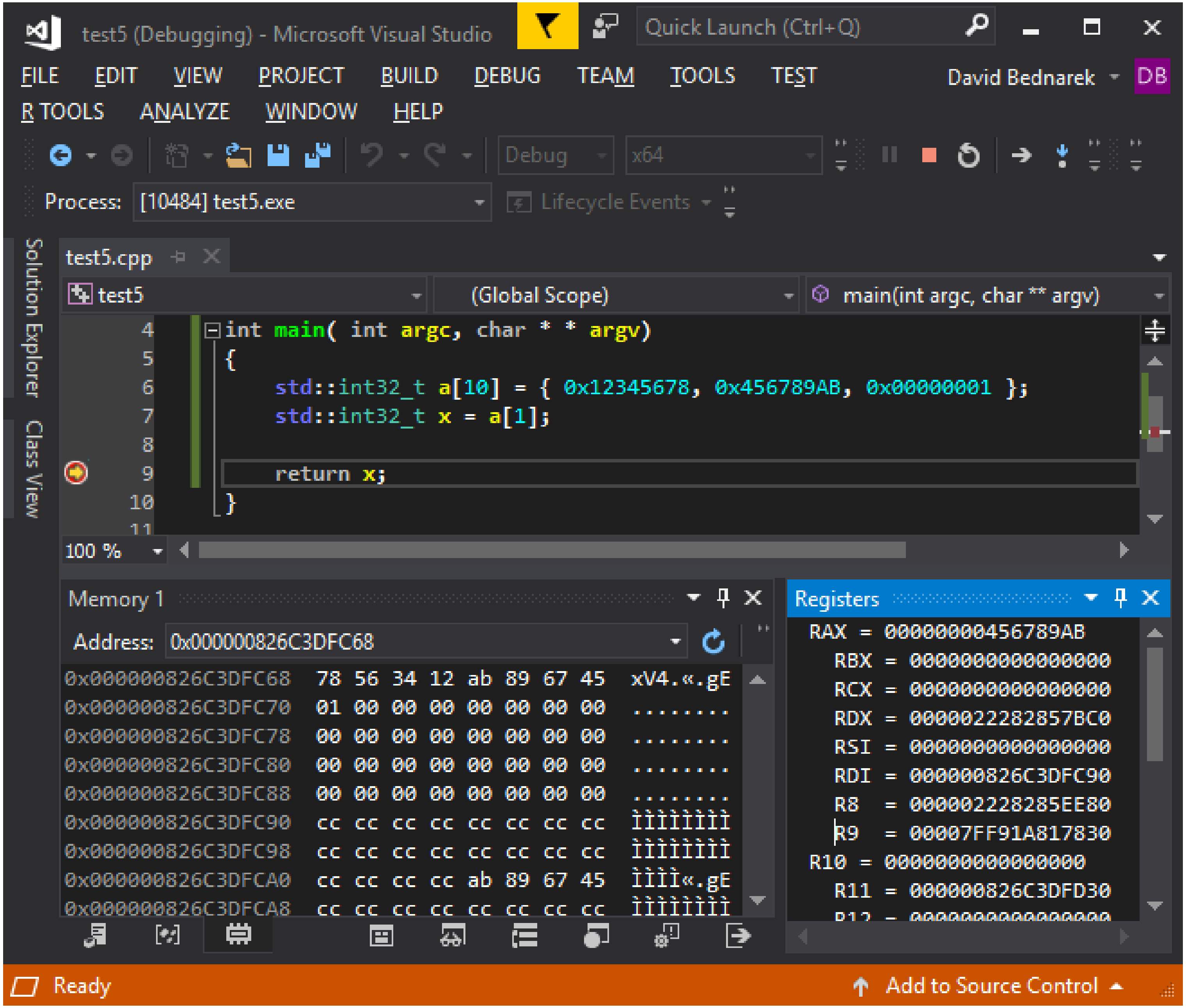


- Registers
  - Fastest storage available
    - register access: < 1 clock
    - memory (cache hit): ~ 4 clocks
    - memory (cache miss): ~ 120 clocks
  - Intel/AMD (64-bit):
    - 15 64-bit integer registers
      - 32/16/8-bit parts accessible
    - 8 80-bit FP registers
    - 16 256-bit vector registers [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]

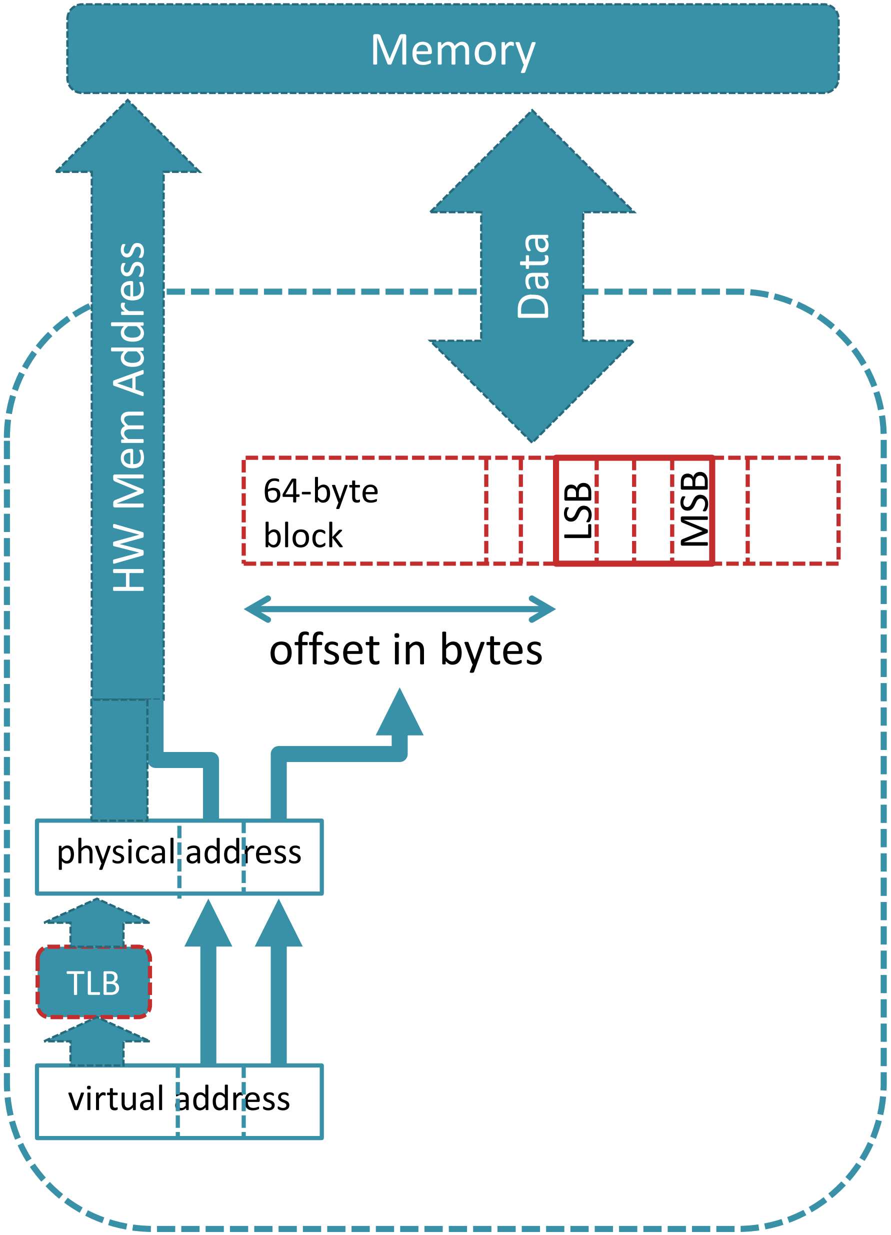


- Drawing conventions
  - Numbers are always written with most-significant-bit on the left
    - Convention since ~ 4th century BC (for Indian decimal numerals)
    - Read by humans starting with MSB - big-endian – consistent with left-to-right writing order
    - Registers are drawn in this order
  - 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

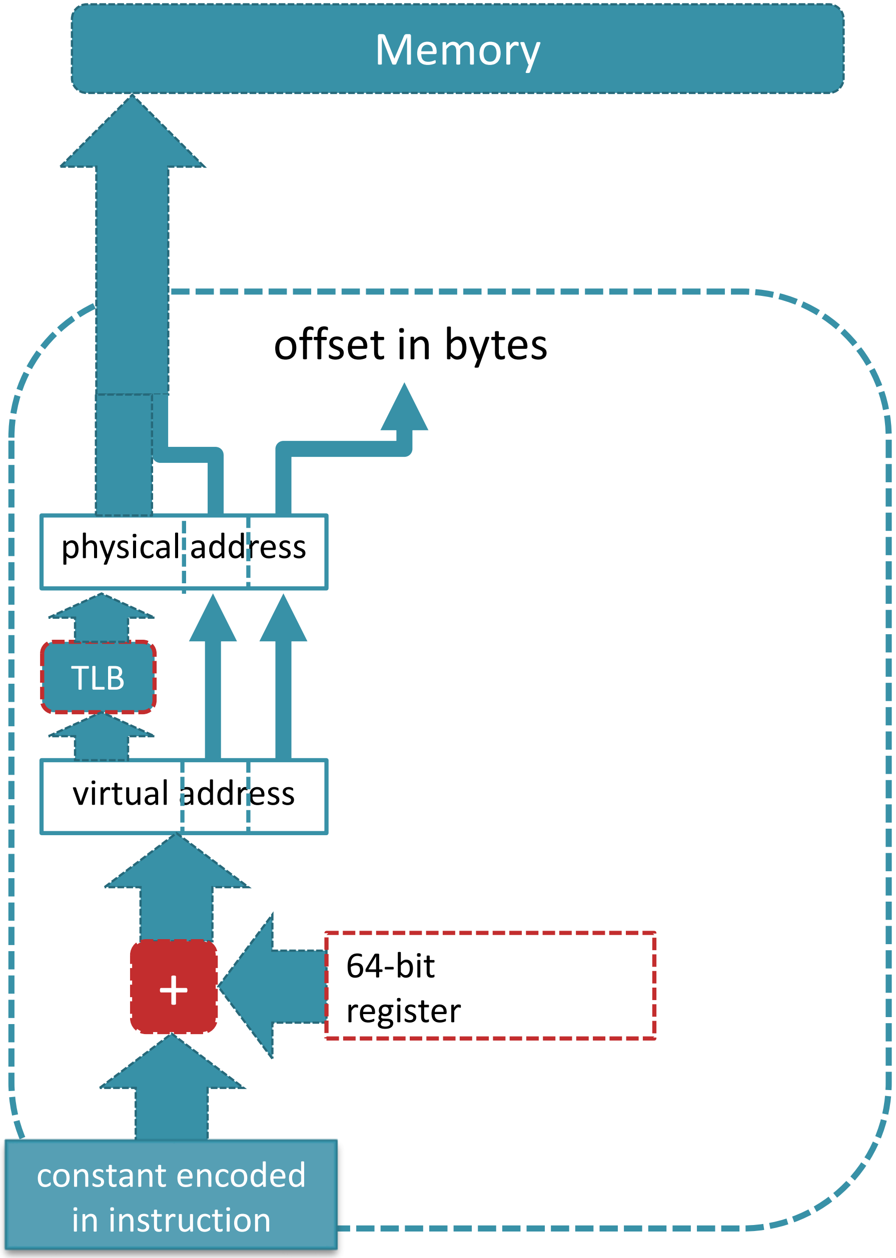


Inside CPU – virtual memory



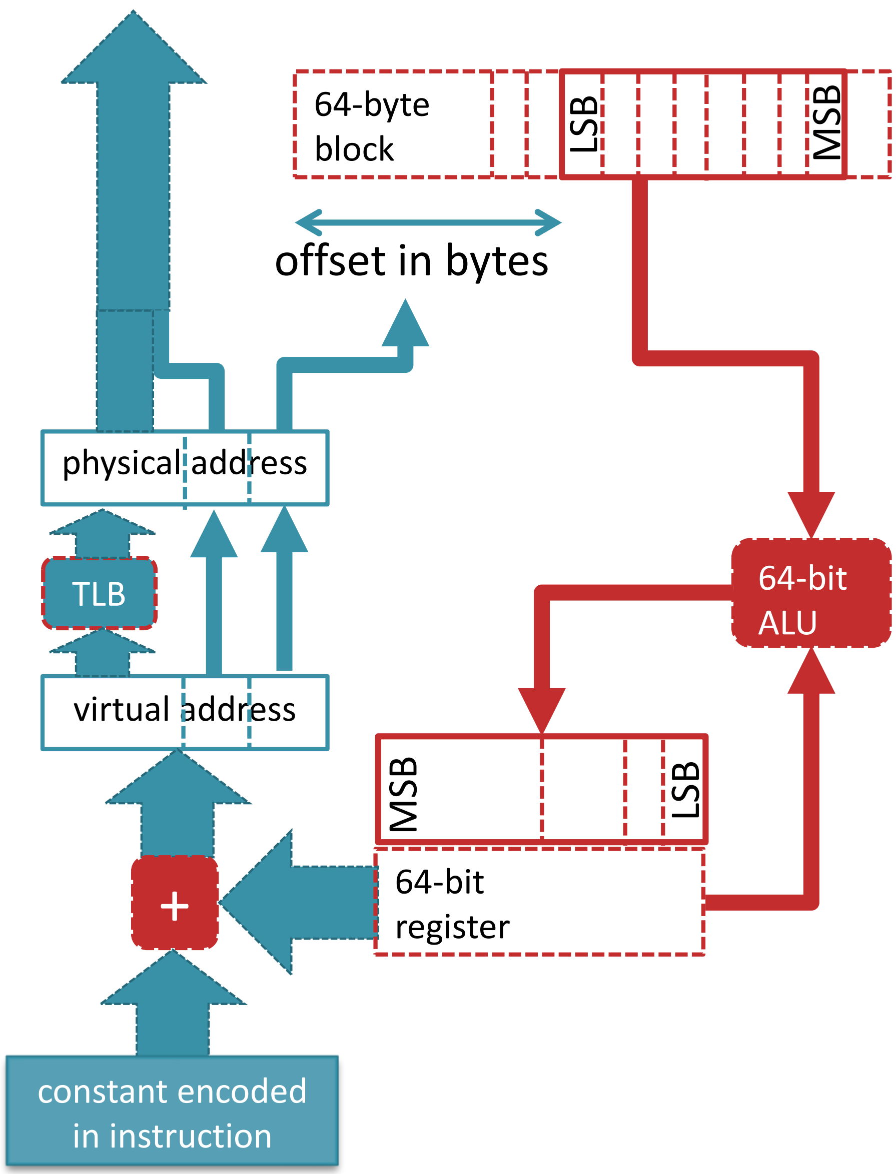
- TLB
  - Translation Look-aside Buffer
  - Translates upper part of addresses
  - Fast hardware mechanism
    - Associative memory
- Address not in TLB
    - or marked as protected
  - Either solved by *page-walk*
    - Slower hardware mechanism
  - Or causes *page-fault*
    - Invoke OS
- Page-fault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

Inside CPU – where addresses come from



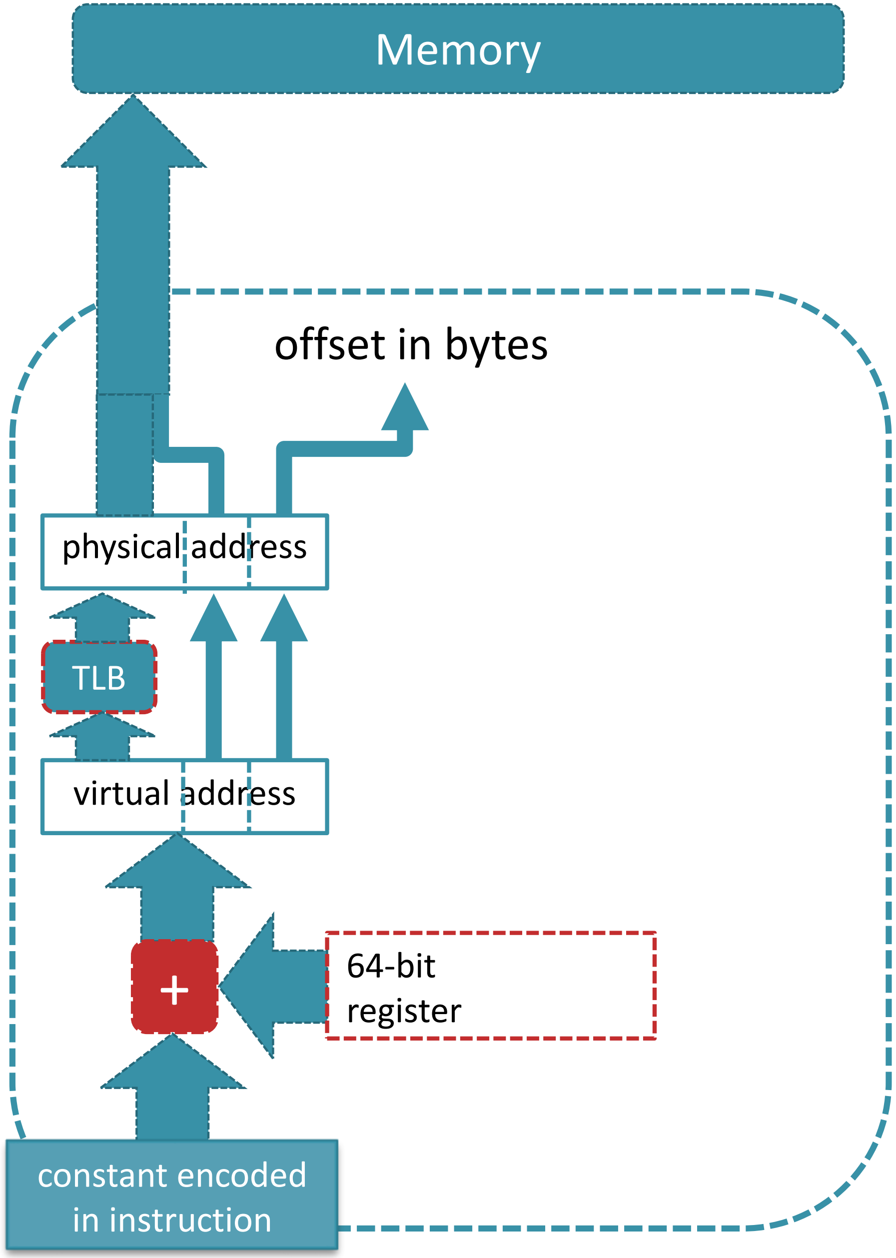
- Virtual address
  - 64-bit integer in most cases
    - Some upper bits ignored
  - 32-bit in 32-bit modes/CPUs
  - Generated as specified in the read/write instruction
  - Instruction encoding allows several *addressing modes*
  - Typical mode: register+constant
    - register name and constant value 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 (Intel/AMD, ARMv8)
  - 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

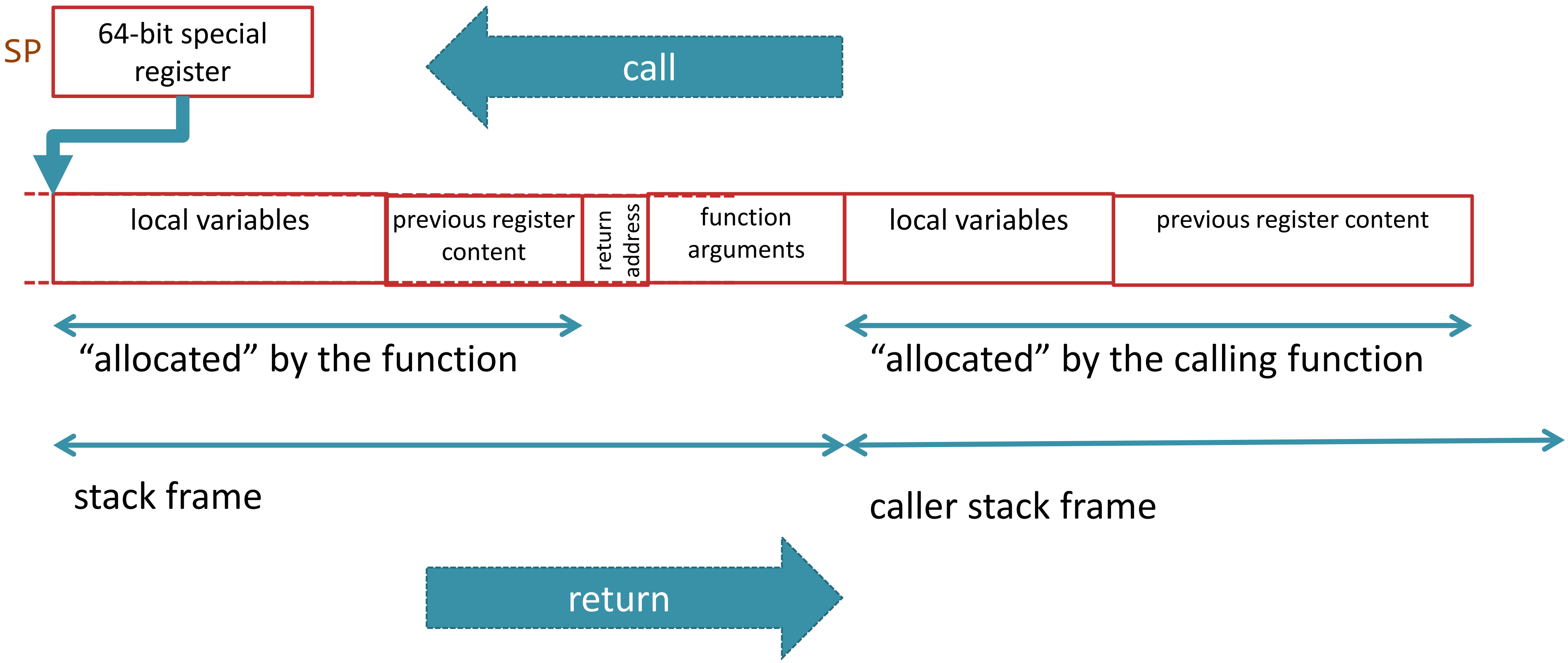
- Where the data reside?
  - Static storage
    - Global, static member, static local variables, string constants
      - One instance (per process)
    - Allocated by compiler/linker/loader (represented in .obj/.dll/.exe files)
  - Thread-local storage
    - C++11 Variables marked "thread_local"
      - 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 – stack frames

- 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

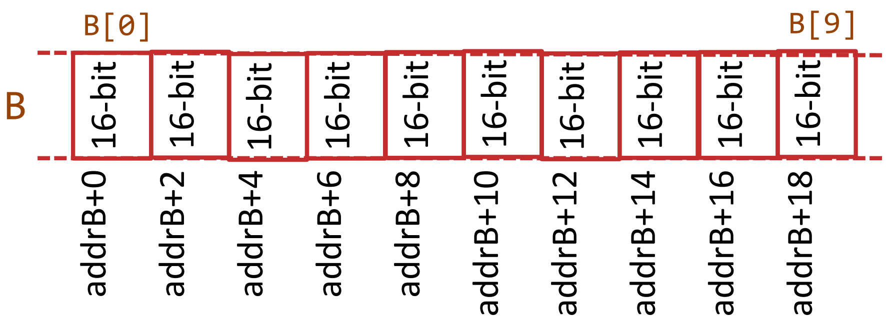
  - Scalar variables (elementary types, decomposed structures)
    - Preferably in registers
      - Previous contents of registers must be saved to the stack when entering the function and restored on exit
    - The rest in stack
  - Aggregate variables (arrays, non-decomposed structures)
    - Must be 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

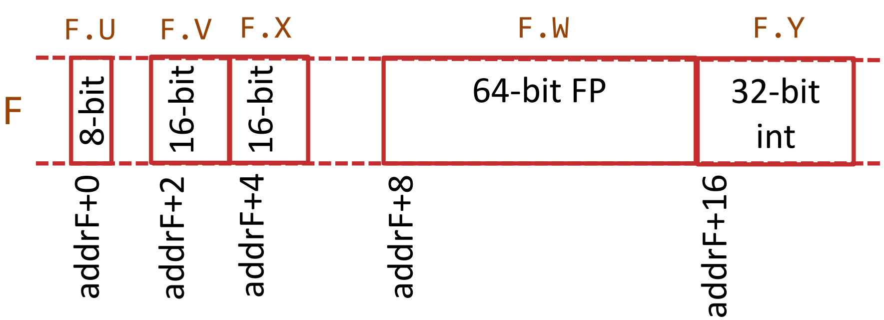
Data in memory – stack frames



- 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 – arrays and structures in C++





- 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

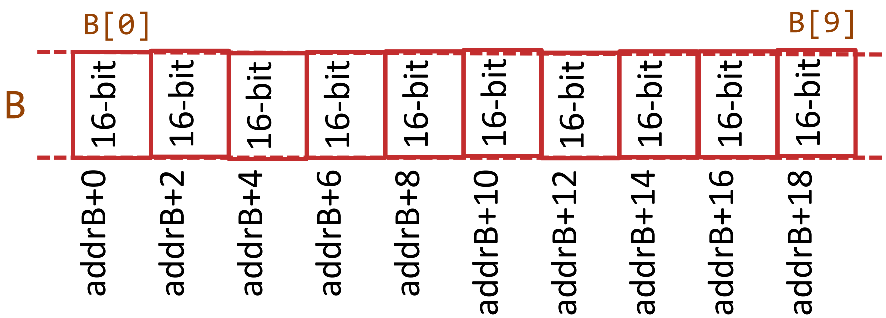
- Structures
  - If F is static (global) variable

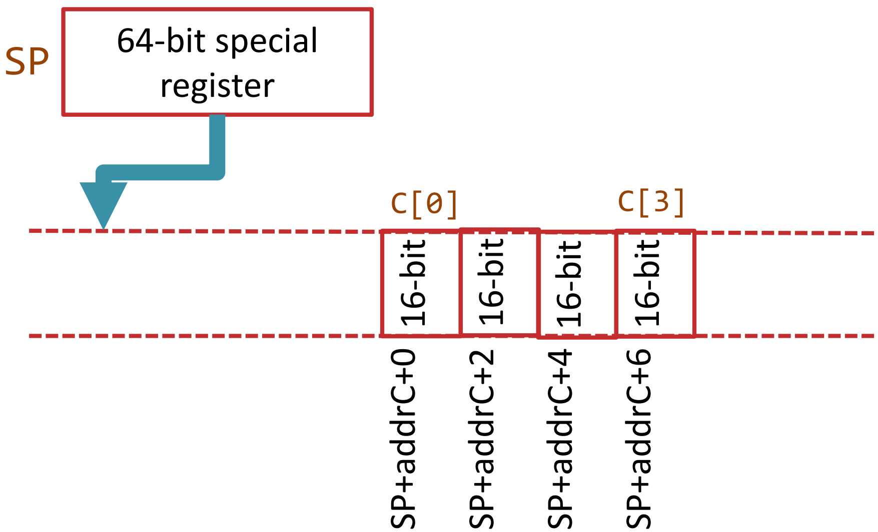
```
struct S { /*...*/ }; static S F;
y = F.V;
```

- Translated to something like

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

- address computed by compiler





- *Static storage* (static/global) variable

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

  - Variable resides at fixed place

```
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];
```

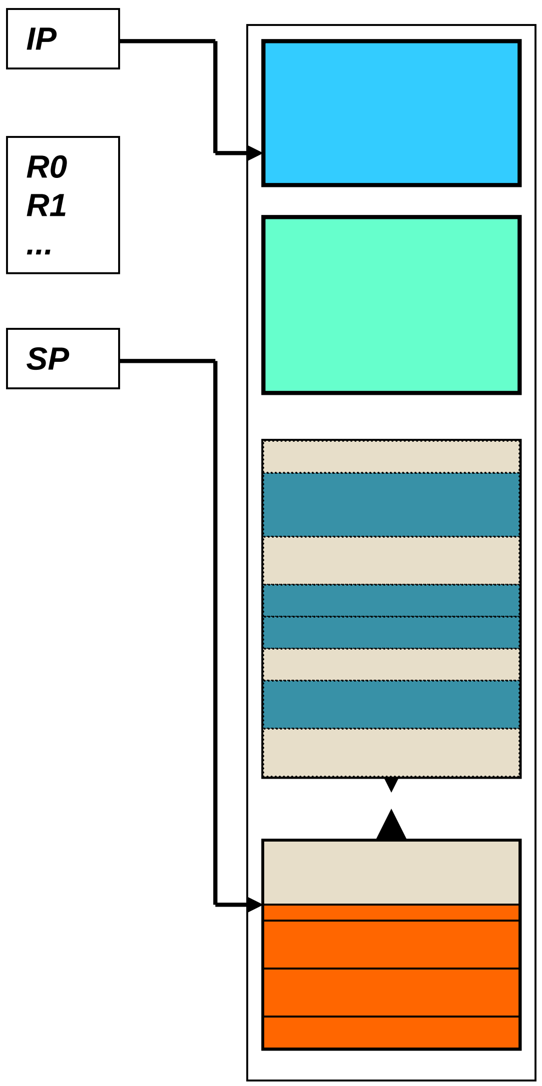
  - The size of array must be determined by the compiler
  - The variable is *the array*

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

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

```
}
```

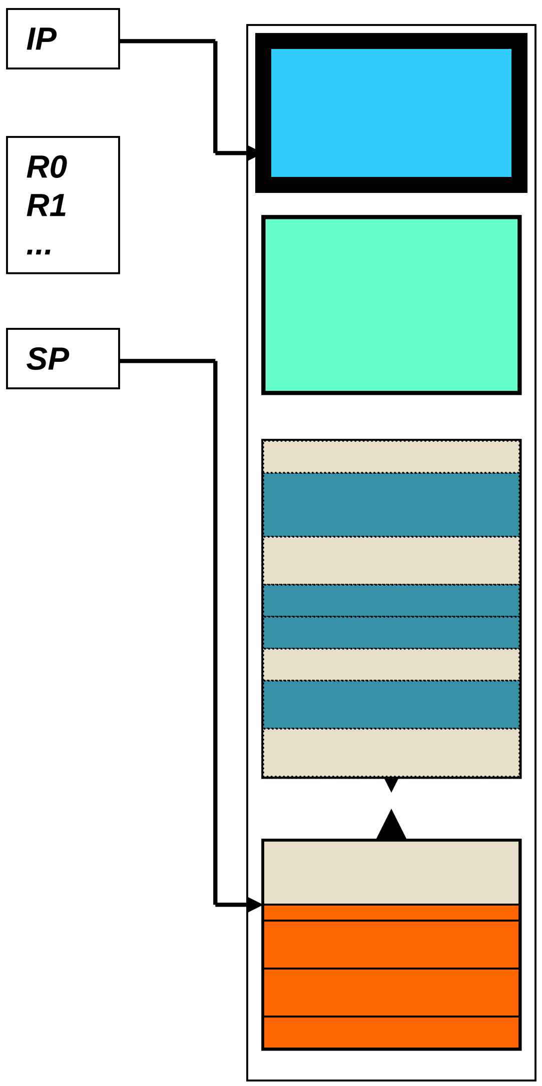
Virtual address space of a process



- Code segment
- Data segment
- Heap
- Stack

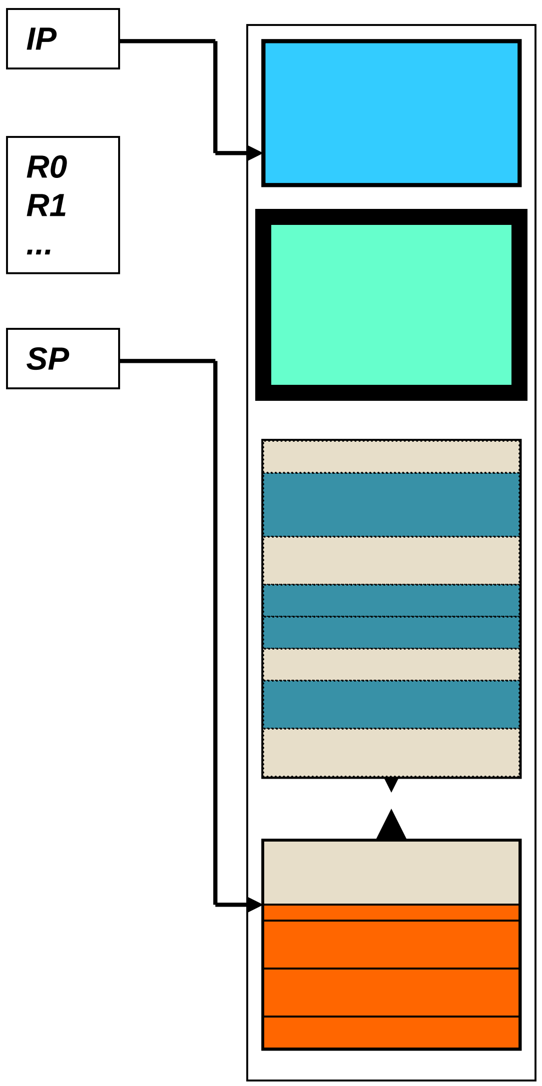
- Segments
  - Regions of memory mapped to the address space by the OS
    - Some described in the executable file
    - Others on request of the process
  - Not necessarily contiguous
    - DLL code shared between processes
    - Heap allocated in batches

Virtual address space of a process



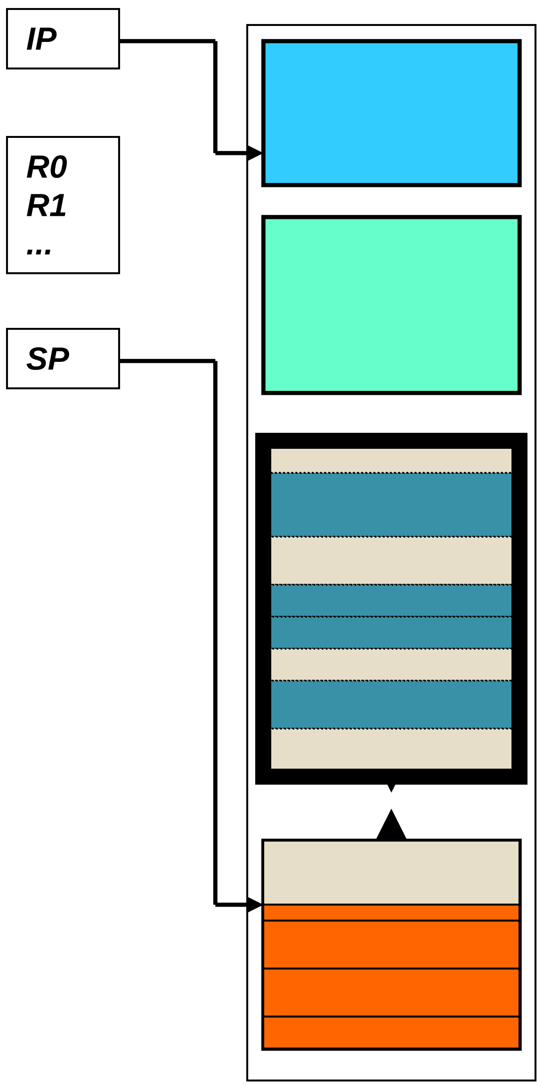
- Code segment
  - Contents prepared by the compiler, part of the executable file
    - Binary code of user/library functions
    - Usually protected against modifications
- Data segment
- Heap
- Stack

Virtual address space of a process



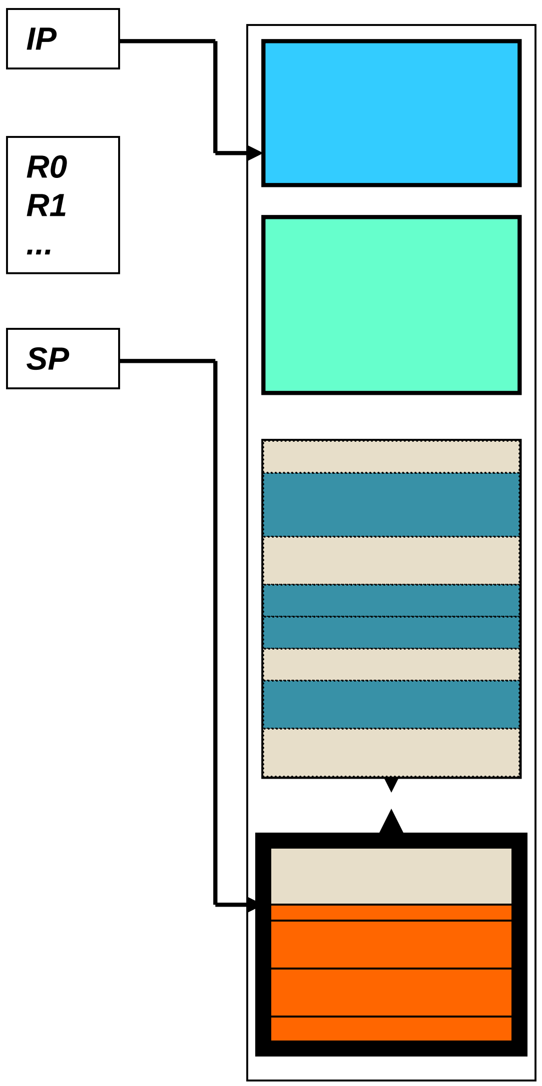
- Code segment
- Data segment
  - Contents prepared by the compiler, part of the executable file
    - Statically allocated data – explicitly or implicitly (by zeros) initialized static/global variables
    - String constants
    - Library data
    - Auxiliary data generated by the compiler (type descriptors, vtables, exception descriptors ...)
- Heap
- Stack

Virtual address space of a process



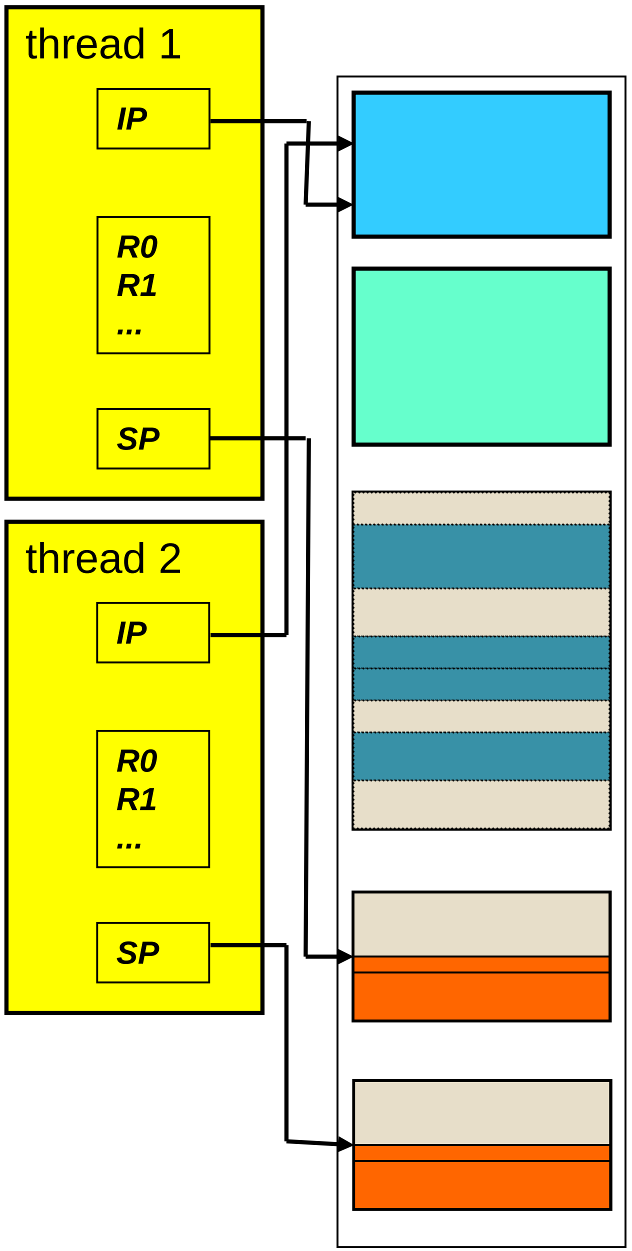
- Code segment
- Data segment
- Heap
  - Created by library code when process is started (before main)
  - Dynamically allocated data
    - C++: new/delete
    - C: malloc/free
  - Occupied blocks of variable sizes + list(s) of unused blocks
    - When there is no unused block of sufficient size, the library may ask the OS for enlarging the heap segment
- Stack

Virtual address space of a process



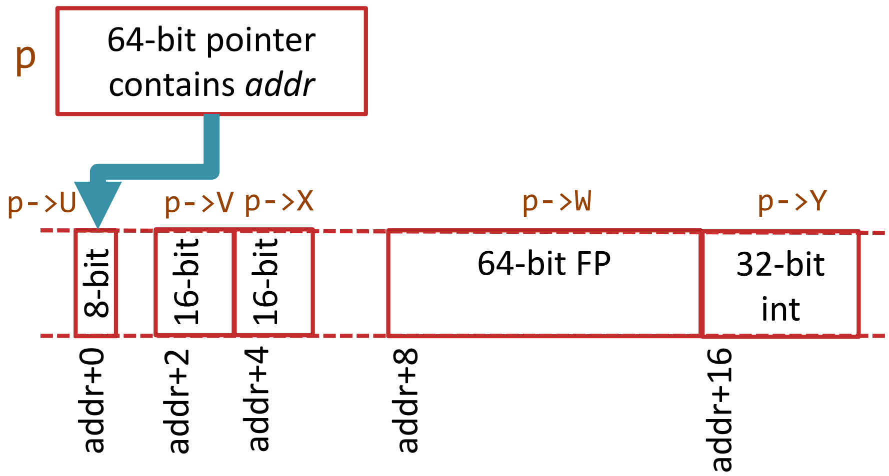
- Code segment
- Data segment
- Heap
- Stack
  - Initialized by OS when process starts
  - Automatic storage (when not placed in registers)
  - Auxiliary data
    - Return addresses
    - Saved registers
  - Multithreaded applications have multiple stacks

Virtual address space of a multi-threaded process



- Each thread owns
  - Instruction pointer
  - Stack pointer
  - Data registers
  - (Thread ID)
- The address space is shared
- Each thread is assigned
  - A stack segment
  - Thread-local storage
    - At stack bottom, or
    - localized by thread-ID
  - May be accessed by other threads using pointers

Data in memory – dynamically allocated structures (before C++11)



- Structure allocated dynamically

```
struct S { /*...*/ };
auto p = new S;
```

  - auto: Type of p is determined by compiler
    - S * = (raw) pointer to S
  - new: (Raw) dynamic allocation
    - library function call (+ constructor call)
  - variable p is assigned some *addr*
    - 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
```

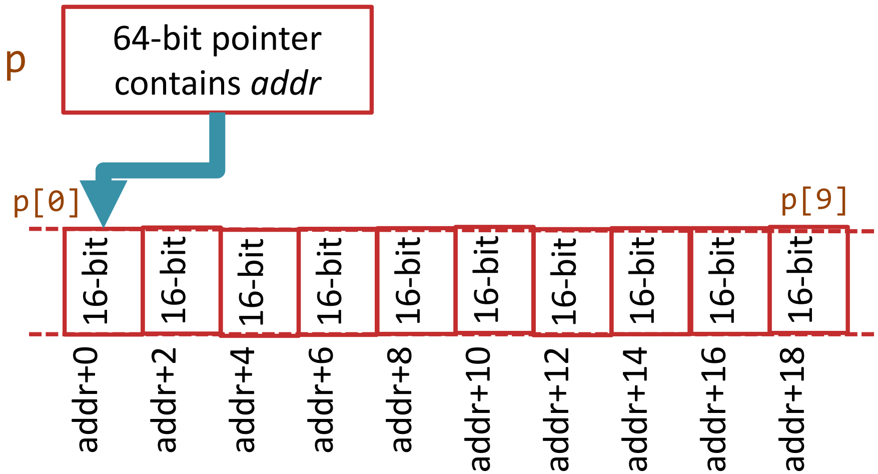
  - note: this is the same memory-read instruction as when reading static array
    - addressing mode: constant + register

- 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 (before C++11)



Array allocated dynamically

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

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

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

- auto: Type of p is determined by compiler
  - std::int16_t * = (raw) pointer to std::int16_t
- 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 is assigned some *addr*
  - 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
```

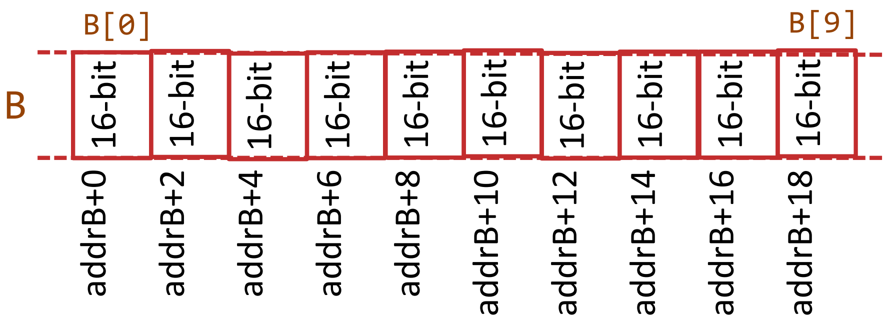
  - 64-bit address computed at runtime

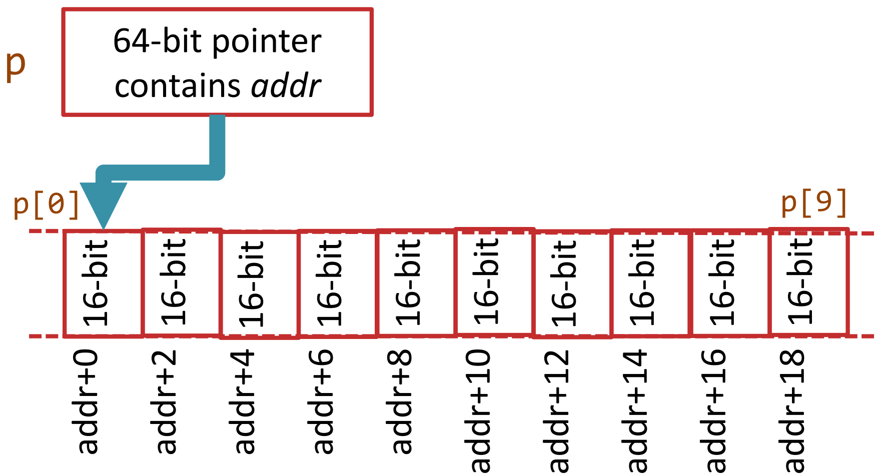
- The block shall be explicitly freed

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

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

Data in memory – dynamically vs statically 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

- Where data reside...
  - Static storage (data segment) – one instance per process

```
T x;                              // global variable
class C { static T x; }           // static member-variable
void f() { static T x; }          // static local variable
```

  - Thread-local storage – one instance per thread (slow – use only if really needed)

```
thread_local T x;                 // thread-local global variable (etc.)
```

  - Automatic storage (stack or CPU register) – one instance per function invocation

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

  - Dynamic allocation (heap)
    - Note: new/delete not directly used in C++11

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

Allowing access temporarily

```
channel ch;

void send_hello()
{
  std::unique_ptr< packet> 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() )
  {
    std::unique_ptr< packet> m = ch.receive();
    std::cout << m->get_contents();
    // the packet is deallocated here
  }
}
```

```
class packet {
  void set_contents( const std::string & s);
  const std::string & get_contents() const;
  /*...*/
};
```

- get_contents returns a reference to data stored inside the packet
  - const prohibits modification
- How long the reference is valid?
  - Probably until modification/destruction of the packet
  - It will last at least during the statement containing the call
    - Provided there is no other action on the packet in the same statement
- set_contents receives a reference to data stored elsewhere
  - const prohibits modification
  - the reference is valid throughout the call

A setter without conversions – various implementations

Before C++11

```
class packet {
public:
  void set_contents(const std::string & s)
  { data_ = s; }
private:
  std::string data_;
};
```

- If the actual is an L-value, there is no better solution

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** from *my_string* to *data_*
  - this operation may recycle the space held previously by *data_*

- If the actual is an R-value, there is unnecessary copying

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents(
   std::move(my_string));
```

  - ***copy*** to *data_*

Simple in C++11

```
class packet {
public:
  void set_contents(std::string s)
  { data_ = std::move( s); }
private:
  std::string data_;
};
```

- If the actual is an L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** from *my_string* to *s*
  - ***move*** from *s* to *data*
  - recycling not possible

- If the actual is an R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents(
   std::move(my_string));
```

  - ***move*** to *s*
  - ***move*** from *s* to *data_*

Full in 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_;
};
```

- If the actual is an L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** from *my_string* to *data_*
    - recycling possible

- If the actual is an R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***move*** to *data_*
    - includes deallocation of the space held previously

A setter without conversions – various implementations

Full in 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_;
};
```

- If the actual is an L-value

```
std::string my_string = /*...*/;
p->set_contents( my_string);
```

  - ***copy*** from *my_string* to *data_*
    - recycling possible
- If the actual is an R-value

```
p->set_contents( "Hello, world!");
p->set_contents( my_string + "!");
p->set_contents( std::move(my_string));
```

  - ***move*** to *data_*
    - includes deallocation of the space held previously

Generic in C++11

```
class packet {
public:
  template< typename X>
  void set_contents(X && s)
  { data_ = std::forward< X>( s); }
private:
  std::string data_;
};
```

- If the actual is of type *std::string*, the behavior is identical to the two-function implementation
  - *std::forward* is a conditional variant of *std::move* for universal references
- If the actual is a different type (e.g. *char[14]*)

```
p->set_contents( "Hello, world!");
```

  - the conversion is done inside the function
  - there may be a specialized *operator=* for this type
    - recycling possible
  - otherwise, the sequence is
    - conversion, including allocation
    - *move to data_*

Declarations and definitions

Declarations and definitions

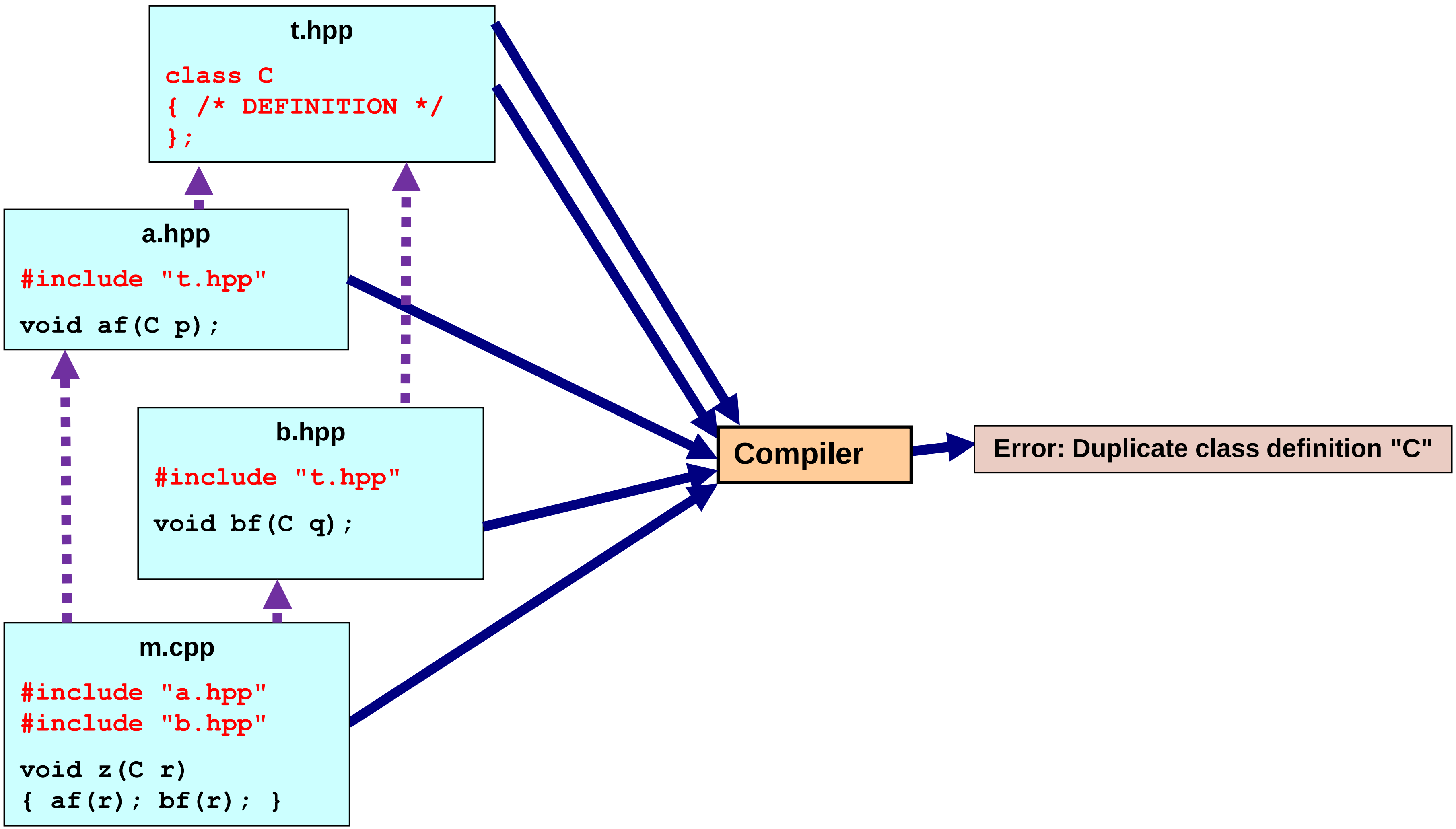
- Declaration
  - A construct to declare the existence (of a class/variable/function/...)
    - Identifier
    - Some basic properties
    - Ensures that (some) references to the identifier may be compiled
      - Some references may require definition
- Definition
  - A construct to completely define (a class/variable/function/...)
    - Class contents, variable initialization, function implementation
    - Ensures that the compiler may generate runtime representation
  - Every definition is a declaration
- Declarations allow (limited) use of identifiers without definition
    - Independent compilation of modules
    - Solving cyclic dependences
    - Minimizing the amount of code that requires (re-)compilation

Declarations and definitions

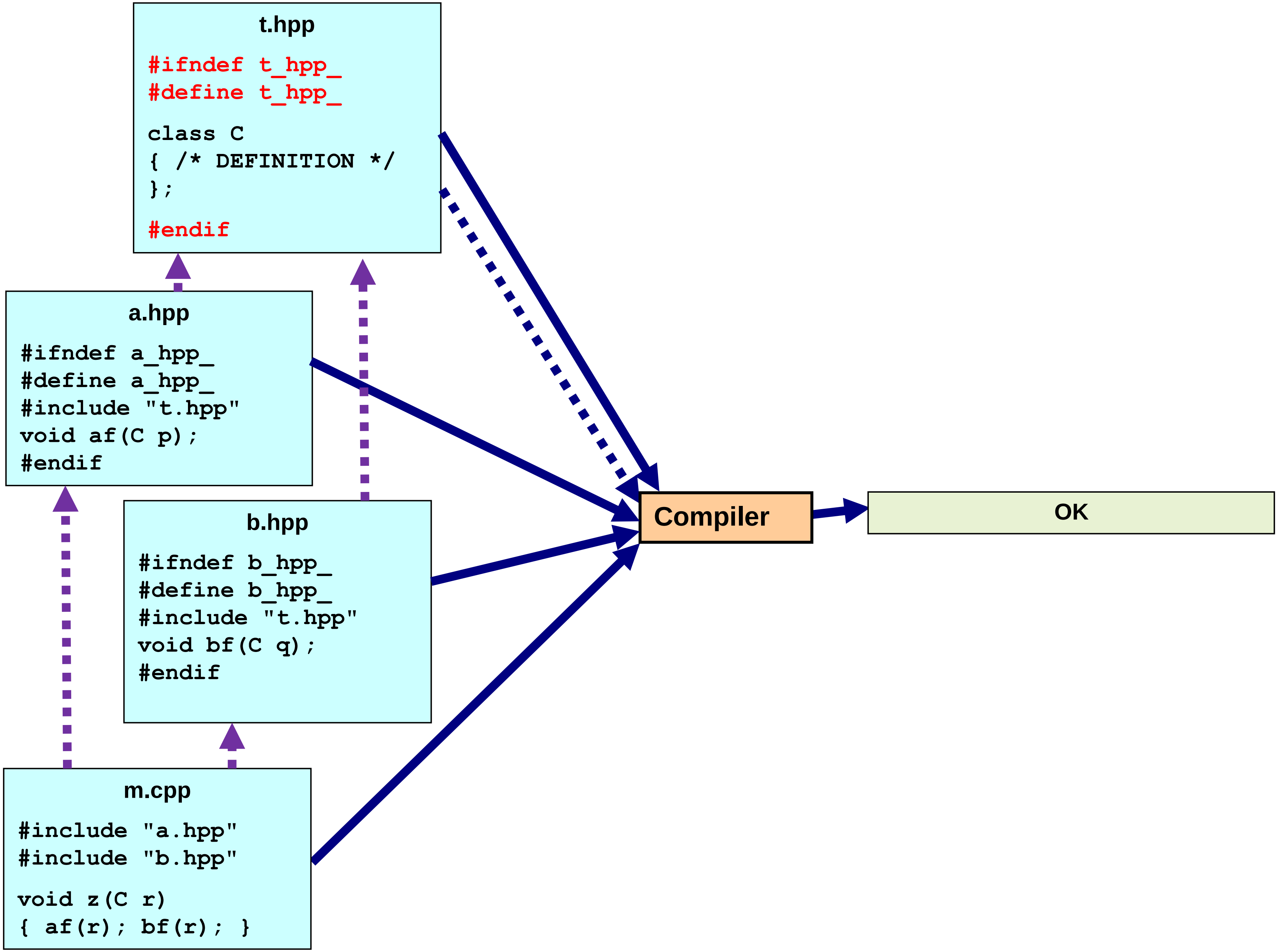
- One-definition rule #1:
  - One *translation unit*...
    - (*module*, i.e. one .cpp file and the .hpp files included from it)
  - ... may contain at most one definition of any item

- One-definition rule #2:
  - Program...
    - (i.e. the .exe file including the linked .dll files)
  - ... may contain at most one definition of a variable or a non-inline function
  - Definitions of classes, types or inline functions may be contained more than once (due to inclusion of the same .hpp file in different modules)
    - If these definitions are not identical, undefined behavior will occur
    - Beware of version mismatch between headers and libraries
  - Diagnostics is usually poor (by linker)

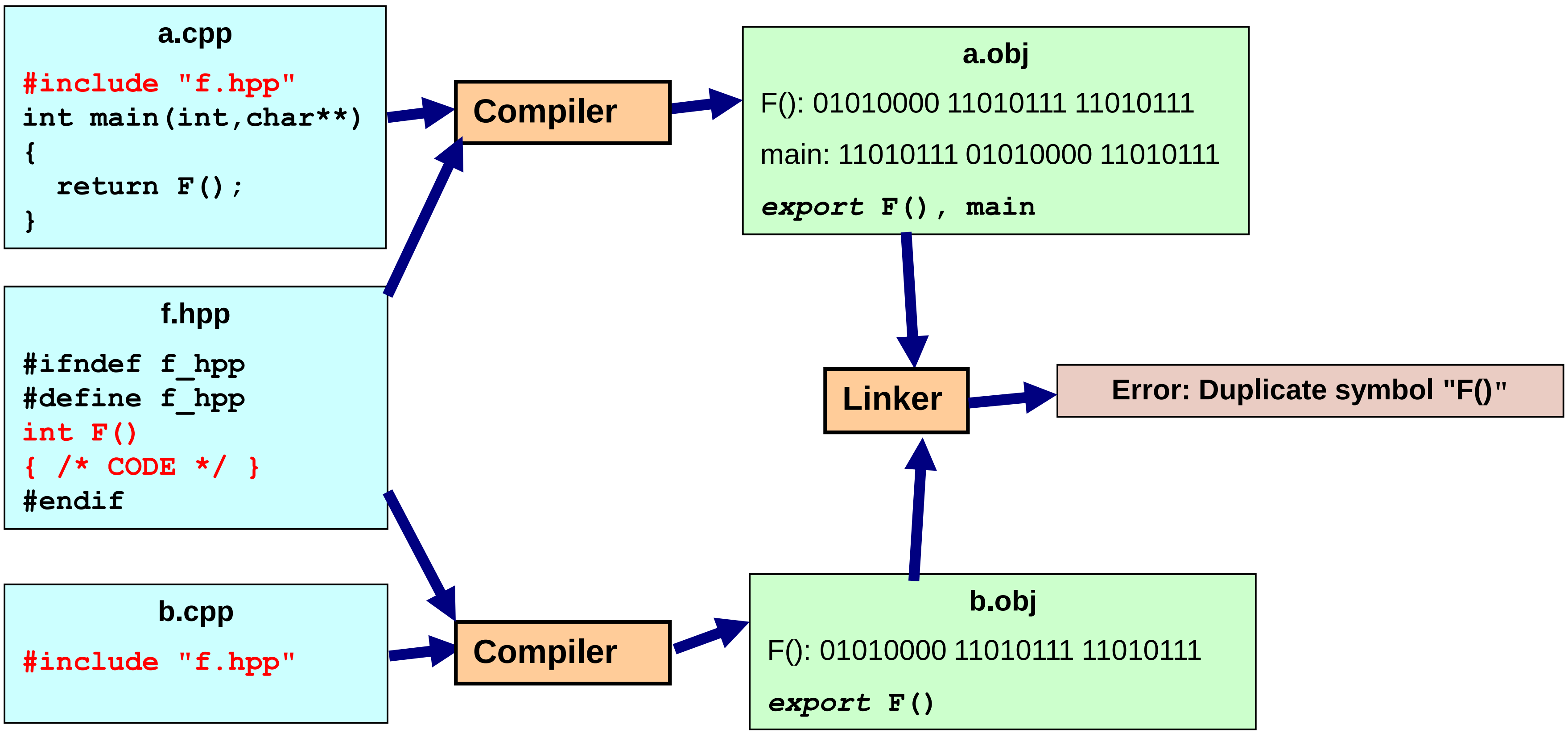
ODR #1 violation



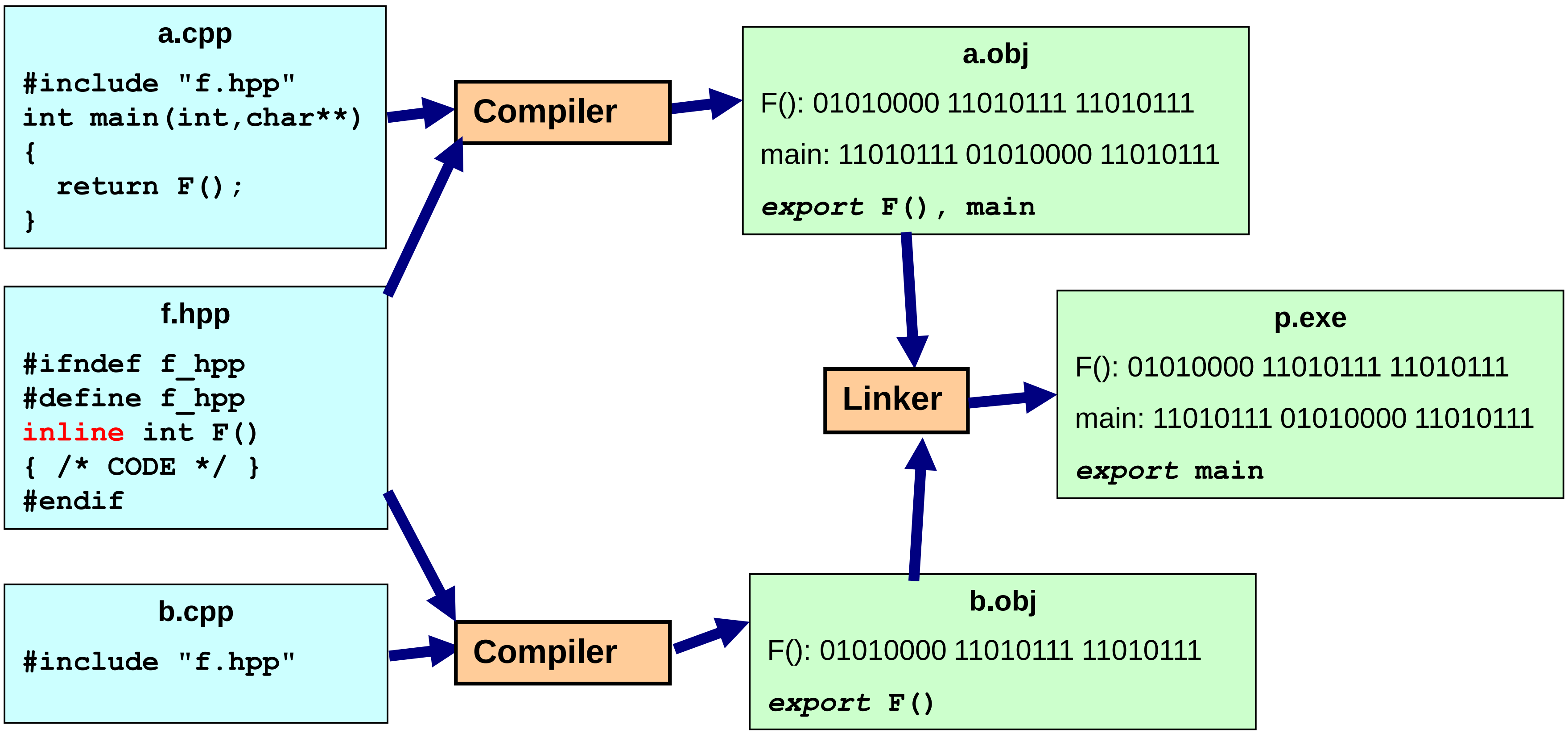
ODR #1 protection



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

Class and type definitions

	Declaration	Definition

Class	`class A;`	`class A {`
		`   ...`
		`};`

Structure (almost equivalent to class)	`struct A;`	`struct A {`
		`   ...`
		`};`

Union (unusable in C++)	`union A;`	`union A {`
		`   ...`
		`};`

Type alias (old style)		`typedef A A2;`
		`typedef A * AP;`
		`typedef std::shared_ptr< A> AS;`
		`typedef A AA[ 10];`
		`typedef A AF();`
		`typedef AF * AFP1;`
		`typedef A (* AFP2)();`
		`typedef std::vector< A> AV;`
		`typedef AV::iterator AVI;`

Type alias (C++11 style)		`using A2 = A;` `using AFP2 = A (*)();`

Function declarations and definitions

non-inline	Declaration (.hpp or .cpp)	Definition (.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	**Declaration (.hpp or .cpp)**	**Definition (.hpp or .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;}`
		`};`

Variable declarations and definitions

	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 member		`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
		`};`

Pointers and references - examples

Returning by reference

- Functions which *compute* their return values must NOT return by reference
  - the computed value usually differs from values of arguments
  - the value of arguments must not be changed
  - there is nothing that the reference might point to

  - Invalid idea #1: Local variable

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

  - RUNTIME ERROR: r disappears during exit from the function
    - before the calling statement can read it

Returning by reference

- Functions which *compute* their return values must NOT return by reference
  - the computed value usually differs from values of arguments
  - the value of arguments must not be changed
  - there is nothing that the reference might point to

  - Invalid idea #2: Dynamic allocation

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

  - PROBLEM: who will deallocate the object?

Returning by reference

- Functions which *compute* their return values must NOT return by reference
  - the computed value usually differs from values of arguments
  - the value of arguments must not be changed
  - there is nothing that the reference might point to

  - Invalid idea #3: Global variable

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

  - PROBLEM: the variable is shared

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

Returning by reference

- Functions which *compute* their return values must return by *value*
  - the computed value usually differs from values of arguments
  - the value of arguments must not be changed
  - there is nothing that a reference might point to

  - (The only) correct function interface:

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

- This body may be shortened to (equivalent by definition):

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

Returning by reference

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

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

- Returning by reference may allow modification of the returned object

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

Returning by reference

- Functions which enable access to existing objects may return by reference
  - Often there are two versions of such function

```
template< typename T, std::size_t N> class array {
public:
```

- Allowing modification of elements of a modifiable container

```
  T & at( std::size_t i)
  { return a_[ i]; }
```

- Read-only access to elements of a read-only container

```
  const T & at( std::size_t i) const
  { return a_[ i]; }

private:
  T a_[ N];
};
void f( array< int, 5> & p, const array< int, 5> & q)
{
  p.at( 1) = p.at( 2);          // non-const version in BOTH cases
  int x = q.at( 3);             // const version
}
```

Templates

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], variable templates [C++14]

```
template< typename T> using array3 = std::array< T, 3>;
template< typename T> constexpr T max = std::numeric_limits<T>::max();
```

Templates

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

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

int maxint = max<int>;
```

  - 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 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
    - resolved via “argument-dependent-lookup” and “overload resolution”

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; }
}
```

Passing template function arguments by value/reference

C++11

- How the arguments are passed in a call to a generic function?

```
std::string a1; std::string & a2 = a1; const std::string & a3 = a1;
f(a1); f(a2); f(a3); f(std::move(a1));
```

  - It depends on the declaration of the function
    - Presence of reference type in the declaration of a2/a3 does NOT matter

```
template< typename T> void f( T x);
```

  - In this case, x is always passed by value

  - If you want pass by reference, always use a “forwarding reference”

```
template< typename T> void f( T && x);
```

    - “Reference collapsing rules” ensure adaptation to both lvalues and rvalues

  - Rationale: plain references cannot adapt

```
template< typename T> void f( T & x);
```

    - In this case, the function **cannot** be called with an rvalue argument

```
f(a1+”.txt”); f(std::move(a1));
```

Forwarding (universal) references

C++11

- Forwarding references may appear
  - as function arguments

```
template< typename T>
void f( T && x)
{
  // ...
}
```

  - as auto variables

```
auto && x = cont.at( i);
```

- Beware, not every T && is a forwarding reference
  - It requires the ability of the compiler to select T according to the actual argument
- The use of reference collapsing tricks is (by definition) limited to T &&
  - The compiler does not try all possible T's that could allow the argument to match
  - Instead, the language defines exact rules for determining T

Forwarding (universal) references

C++11

- In this example, T && is **not** a forwarding reference

```
template< typename T>
class C {
  void f( T && x) {
    // ...
  }
};
C<X> o; X lv;
o.f( lv); // error: cannot bind an rvalue reference to an lvalue
```

- The correct implementation

```
template< typename T>
class C {
  template< typename T2>
  void f( T2 && x) {
    // ...
  }
};
```

Perfect forwarding

- Problem: Forwarding a rvalue/lvalue reference to another function

C++11

```
template< typename T> void f( T && p)
{
  g( p);
}

X lv;
f( lv);
```

- If an lvalue is passed: T = X & and p is of type X &
  - p appears as lvalue of type X in the call to g

```
f( std::move( lv));
```

- If an rvalue is passed: T = X and p is of type X &&
  - p appears as lvalue of type X in the call to g
  - Inefficient – move semantics lost

Perfect forwarding

- Perfect forwarding

C++11

```
template< typename T> void f( T && p)
{
  g( std::forward< T>( p));
}
```

  - std::forward< T> is simply a cast to T &&

```
X lv;
f( lv);
```

- T = X &
  - std::forward< T> returns X & due to reference collapsing
  - The argument to g is an lvalue

```
f( std::move( lv));
```

- T = X
  - std::forward< T> returns X &&
  - The argument to g is an rvalue
  - std::forward< T> acts as std::move in this case

Variadic templates

Variadic templates

- Motivation
  - Allow forwarding of variable number of arguments
    - e.g. in *emplace* functions

C++11

```
template< typename ... TList>
void f( TList && ... plist)
{
  g( std::forward< TList>( plist) ...);
}
```

- Allows many dirty tricks

C++17

```
template< typename ... TList>
void print( TList && ... plist)
{
  (std::cout << ... << plist) << std::endl;
}
```

Variadic templates

C++11

- Template heading
  - Allows variable number of type arguments

```
template< typename ... TList>

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

  - *typename ...* declares named ***template parameter pack***

  - may be combined with regular type/constant arguments

```
template< typename T1, int c2, typename ... TList>

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

  - also in partial template specializations

```
template< typename T1, typename ... TList>

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

Variadic templates

C++11

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

- template parameter pack - a list of types
- may be referenced inside the template:
  - always using the suffix ...

- as type arguments to another template:

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

- in argument list of a function declaration:

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

  - this creates a named function parameter pack

- in several less frequent cases, including
  - base-class list:

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

  - number of elements in the parameter pack:

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

Variadic templates

C++11

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

- named ***function parameter pack***
- may be referenced inside the function:
  - always using the suffix ...
  - as parameters in a function call or object creation:

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

- constructor initialization section (when variadic base-class list is used)

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

- other infrequent cases

Variadic templates

C++11

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

- parameter packs may be wrapped into a type construction/expression
  - the suffix ... works as compile-time "for_each"
  - parameter pack name denotes the place where every member will be placed
    - more than one pack name may be used inside the same ... (same length required)

- the result is
  - a list of types (in a template instantiation or a function parameter pack declaration)

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

  - a list of expressions (in a function call or object initialization)

```
  g( make_pair( 1, & plist) ...);
  h( static_cast< TList *>( plist) ...);      // two pack names in one ...
  m( sizeof( TList) ...);                     // different from sizeof...( TList)
```

  - other infrequent cases

fold expressions - variadic templates

- C++14 – recursion

C++14

```
auto old_sum(){ return 0; }
template<typename T1, typename... T>
auto old_sum(T1 s, T... ts){ return s + old_sum(ts...); }
```

- C++17 - simplification

```
template<typename... T> ????( T... pack)
( pack op ... )                // P1 op (P2 op ... (Pn-1 op Pn))
( ... op pack )                // ((P1 op P2) op ... Pn-1) op Pn
( pack op ... op init )        // P1 op (P2 op ... (Pn op init))
( init op ... op pack )        // ((init op P1) op ... Pn-1) op Pn
```

```
template<typename... T>
auto fold_sum(T... s){
    return (... + s);
}
```

C++17

```
template<typename... T>
auto fold_sum1(T... s){
    return (0 + ... + s);
}
```

```
template<typename ...Args> void prt(Args&&... args)
{ (cout << ... << args) << '\n';  }
```

the std::tuple template

C++11: <utility>

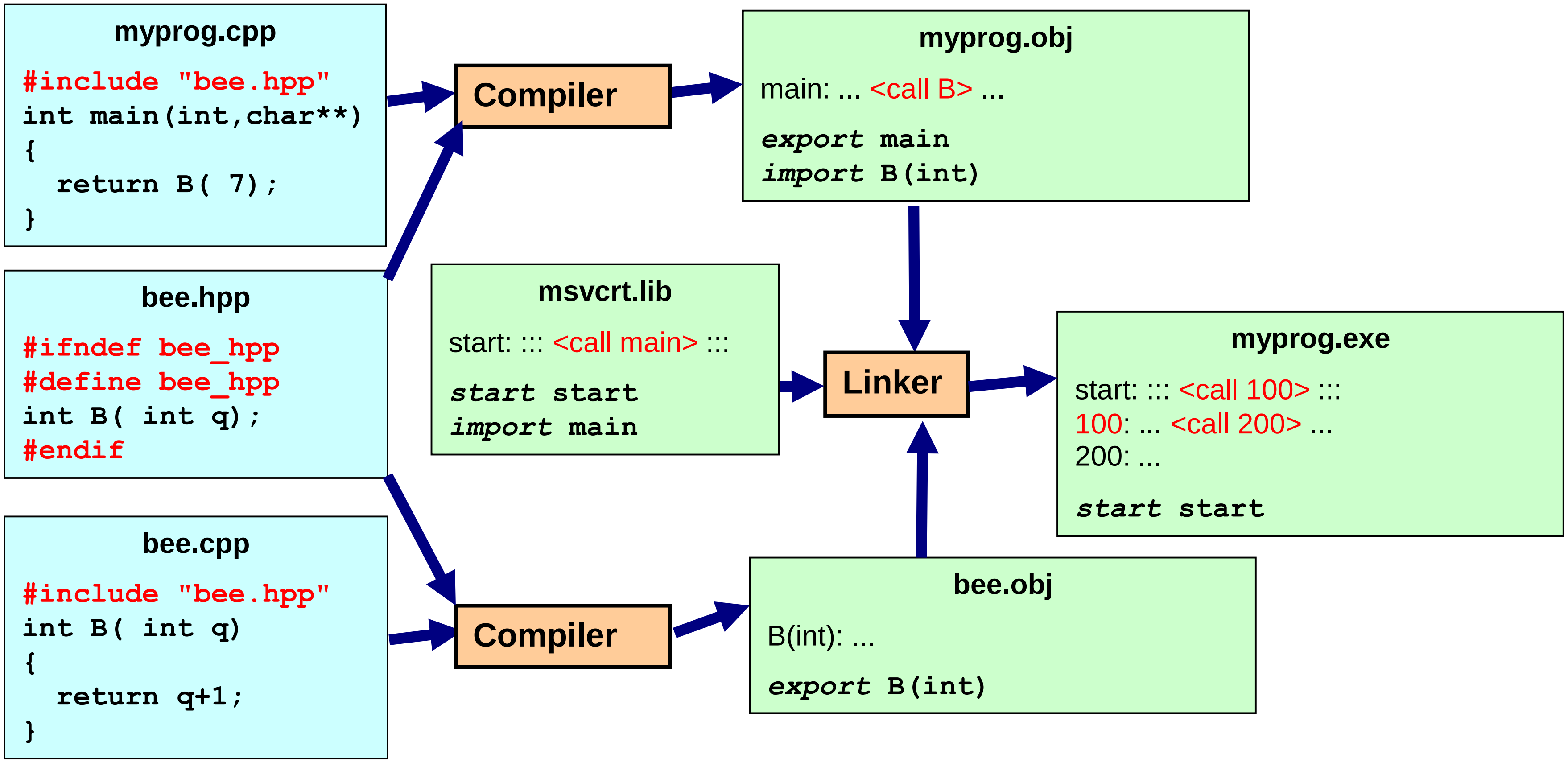
```
template <class ... Types> class tuple {
public:
  tuple( const Types & ...);
  /* black magic */
};
template < size_t I, class T> class tuple_element {
public:
  using type = /* black magic */;
};
template < size_t I, class ... Types>
  typename tuple_element< I, tuple< Types ...> >::type &
  get(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

- example

```
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);
```

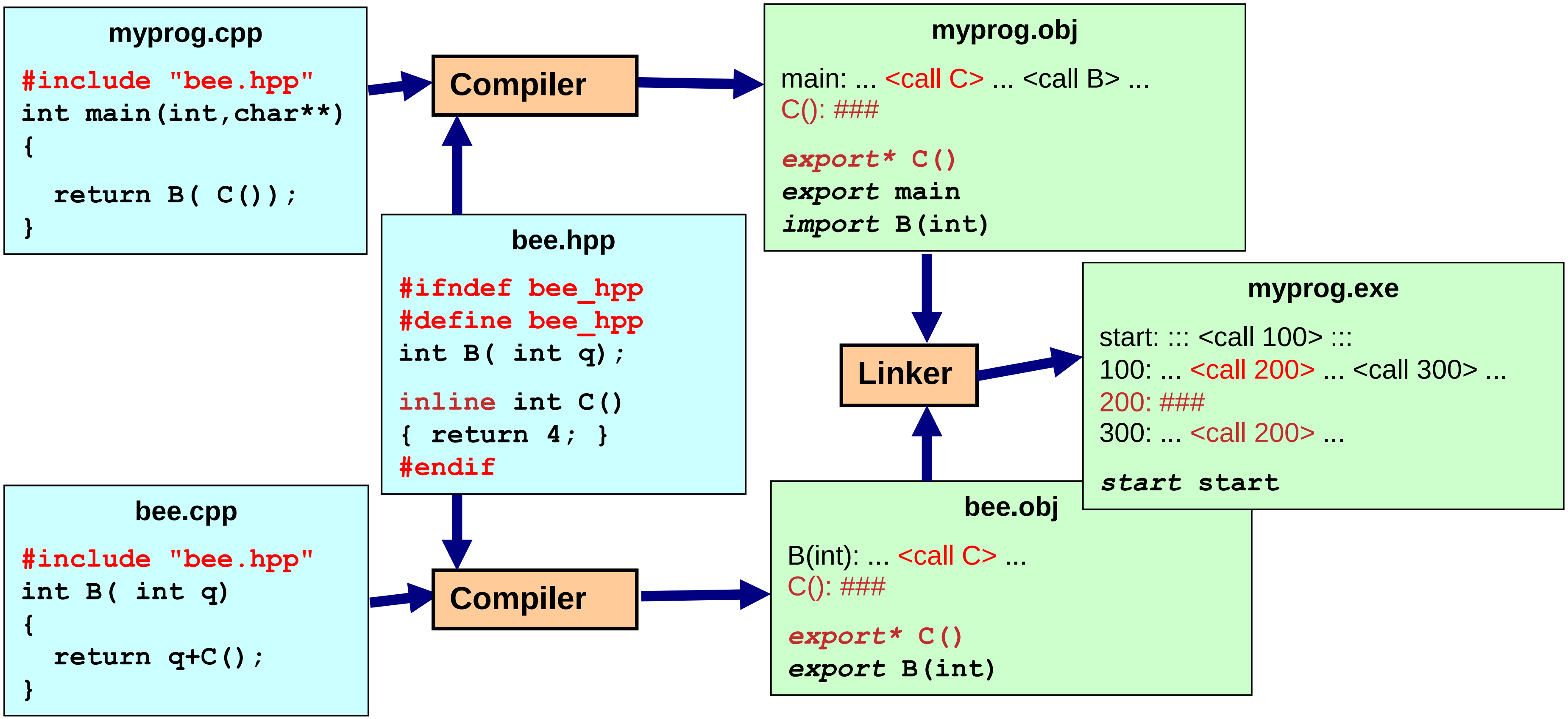
Compilation and linking

Independent compilation of modules (old style)



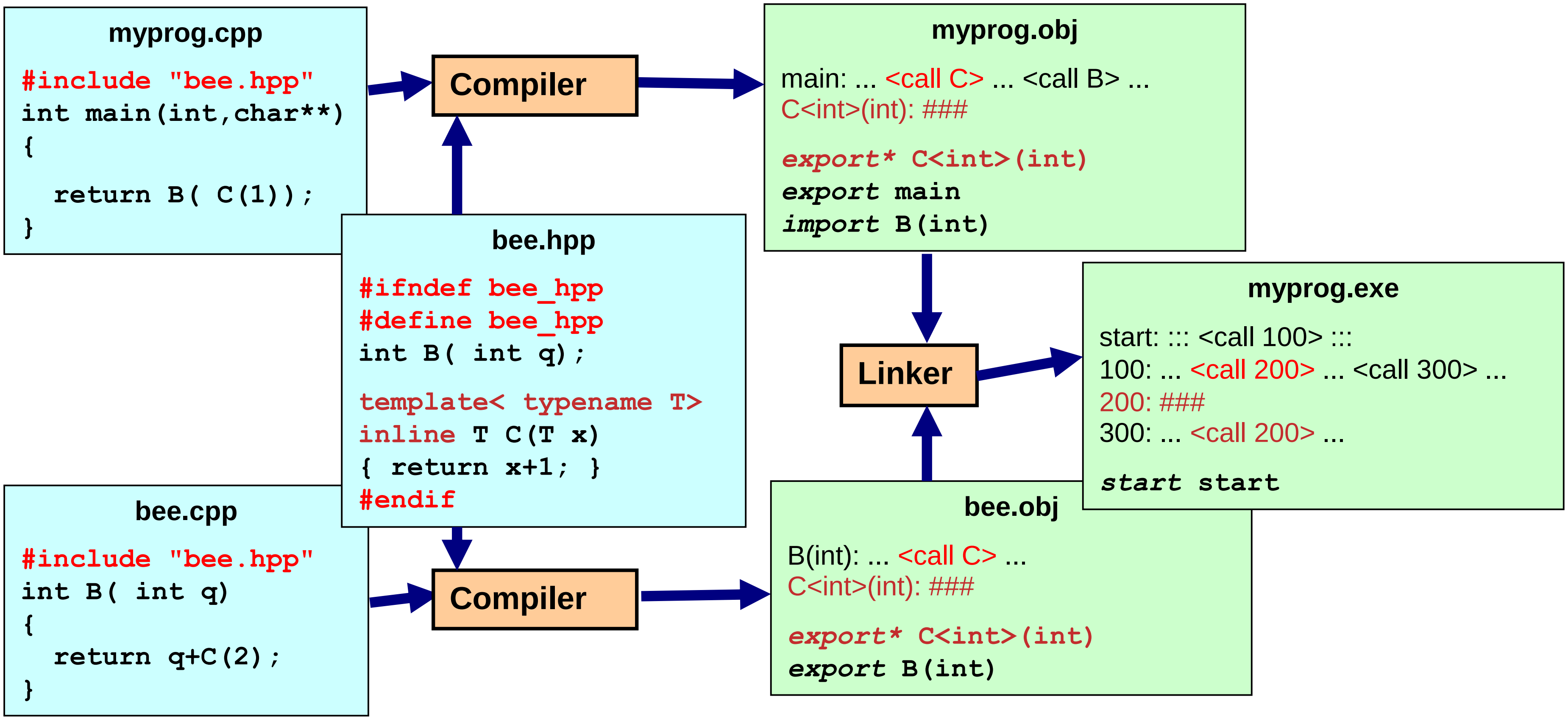
- Non-inline function B defined in bee.cpp, called from myprog.cpp
  - Declaration of B shared in bee.hpp
- Body of B is compiled and optimized only once
- Old style compilers produce binary code of target CPU

Inline functions (old style)



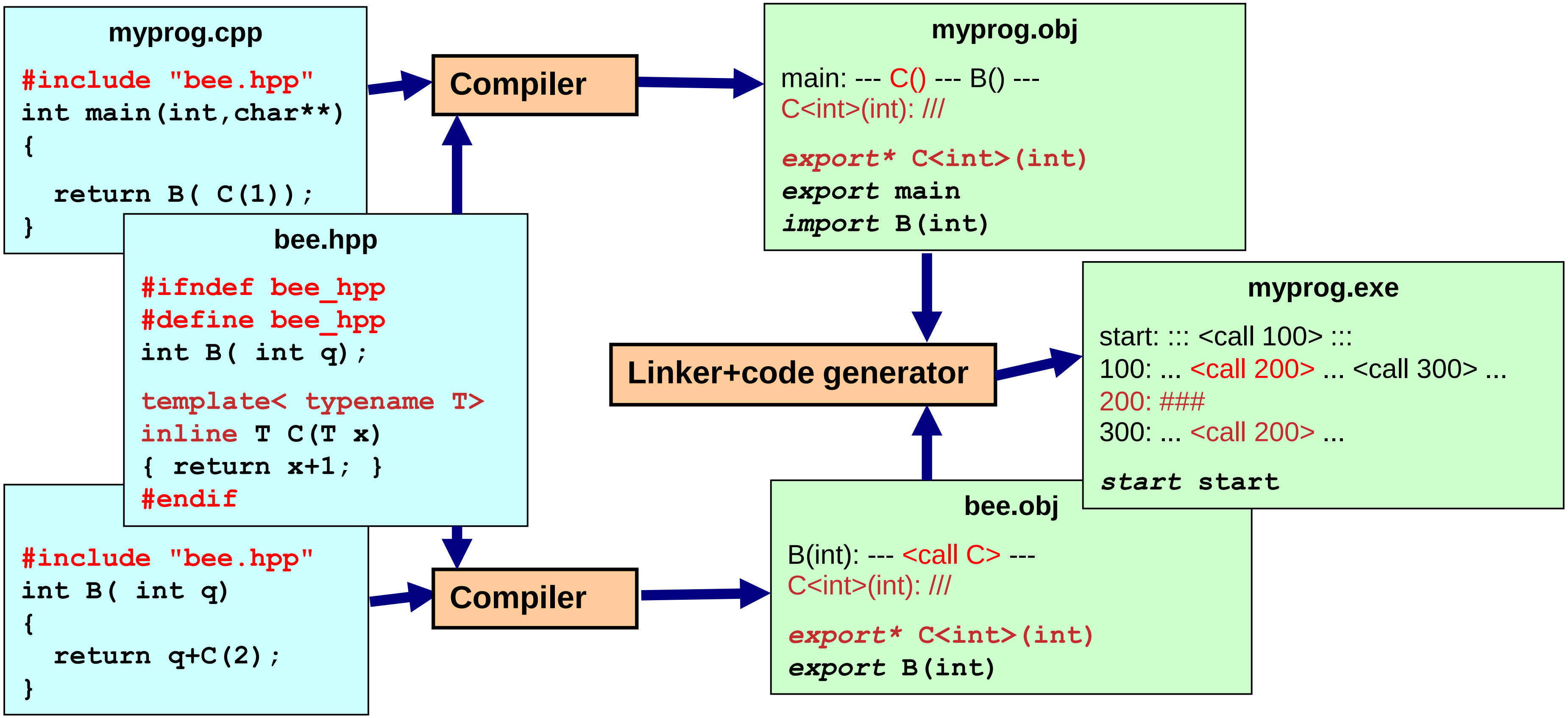
- Inline function C defined in bee.hpp, called from myprog.cpp and bee.cpp
- Body of C is compiled and optimized twice
  - The compiler *may* place the function body instead of the call (inlining aka procedure integration)
    - The compiler may do inlining even if the function is not marked as inline
  - 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)



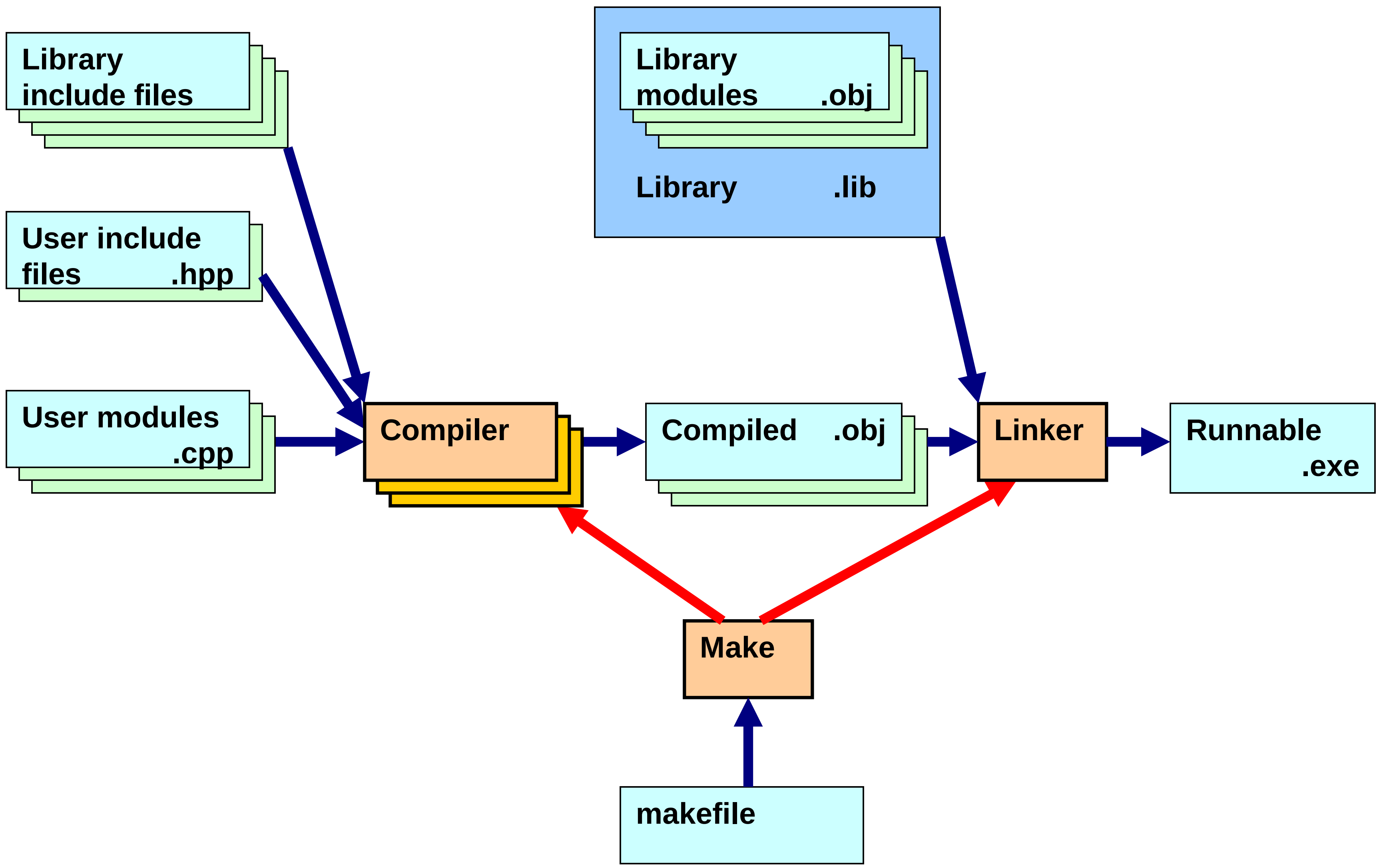
- Template inline function C defined in bee.hpp, called from myprog.cpp and bee.cpp
- Body of C is instantiated as C<int>, compiled and optimized twice
- The body of C must be visible for the compiler which does the instantiation
  - Otherwise, there will be no compiler to instantiate it
- All template 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)



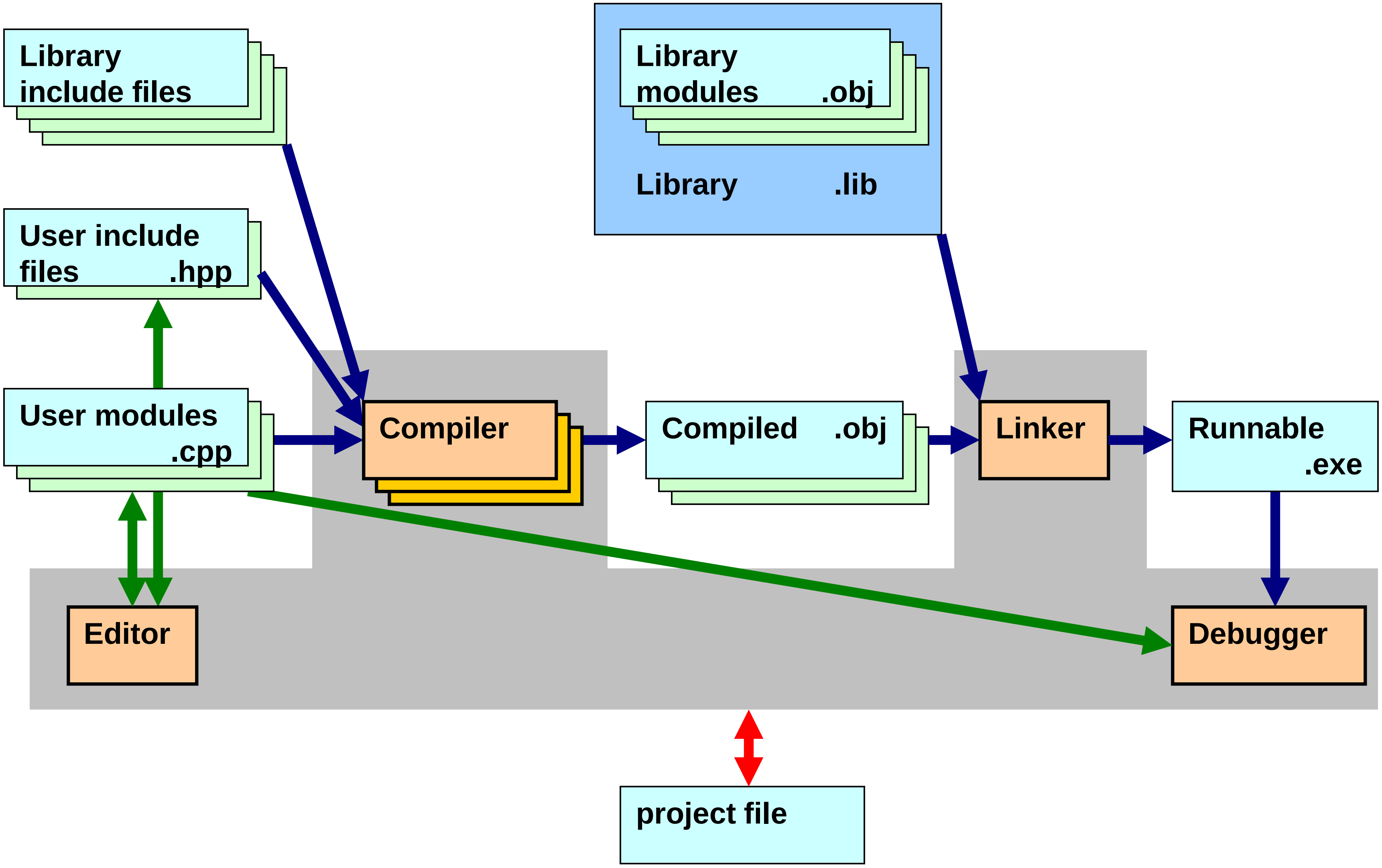
- Modern approach to compiling and linking
  - Function bodies are compiled only to some intermediate code
    - Object modules usually contain no binary code of the target platform (but they still can)
  - Templates are instantiated multiple times
    - Template code must still be located in header 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

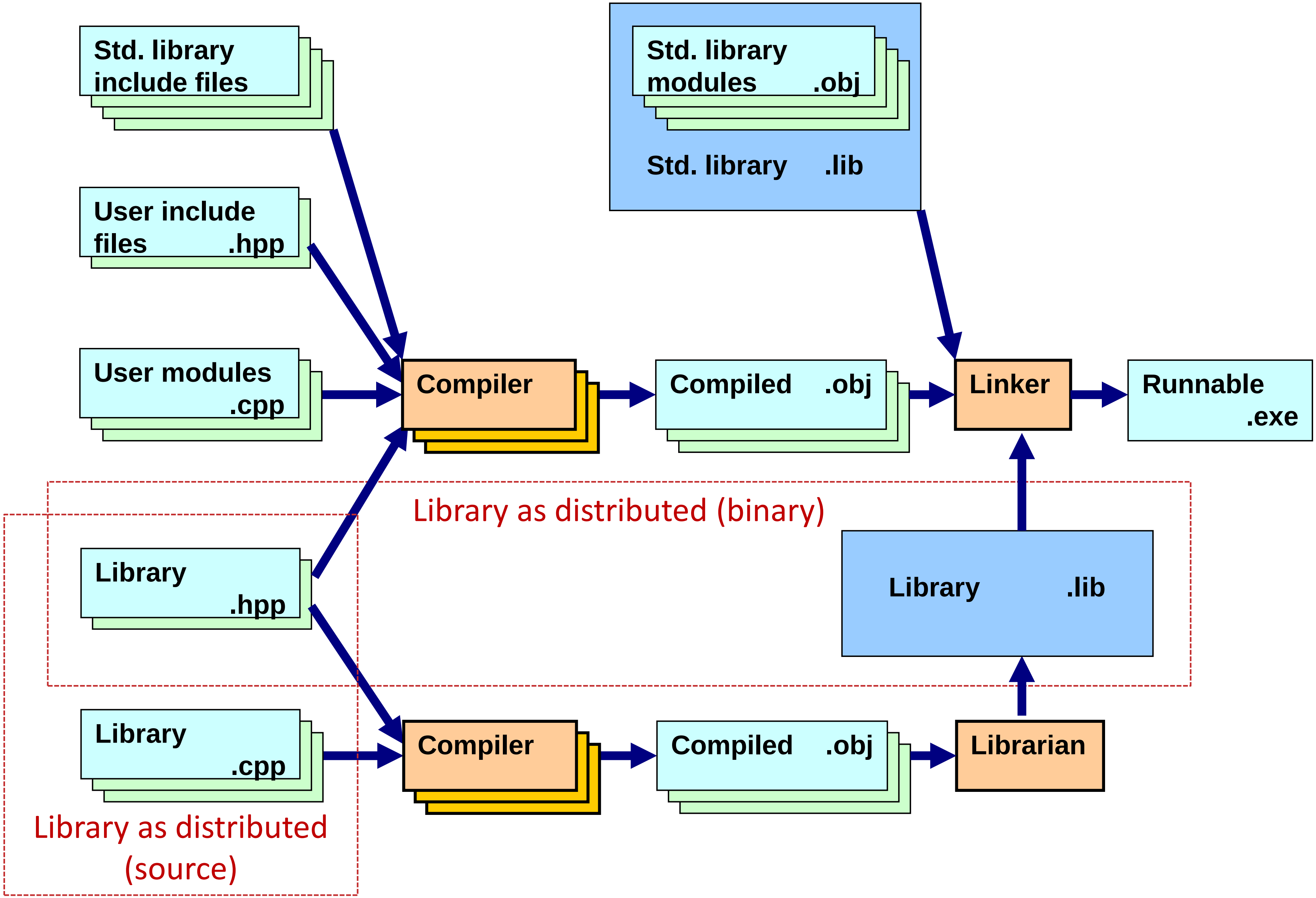


Library include files
User include files .hpp
User modules .cpp
Library modules .obj
Library .lib
Compiler
Compiled .obj
Linker
Runnable .exe
Make
makefile

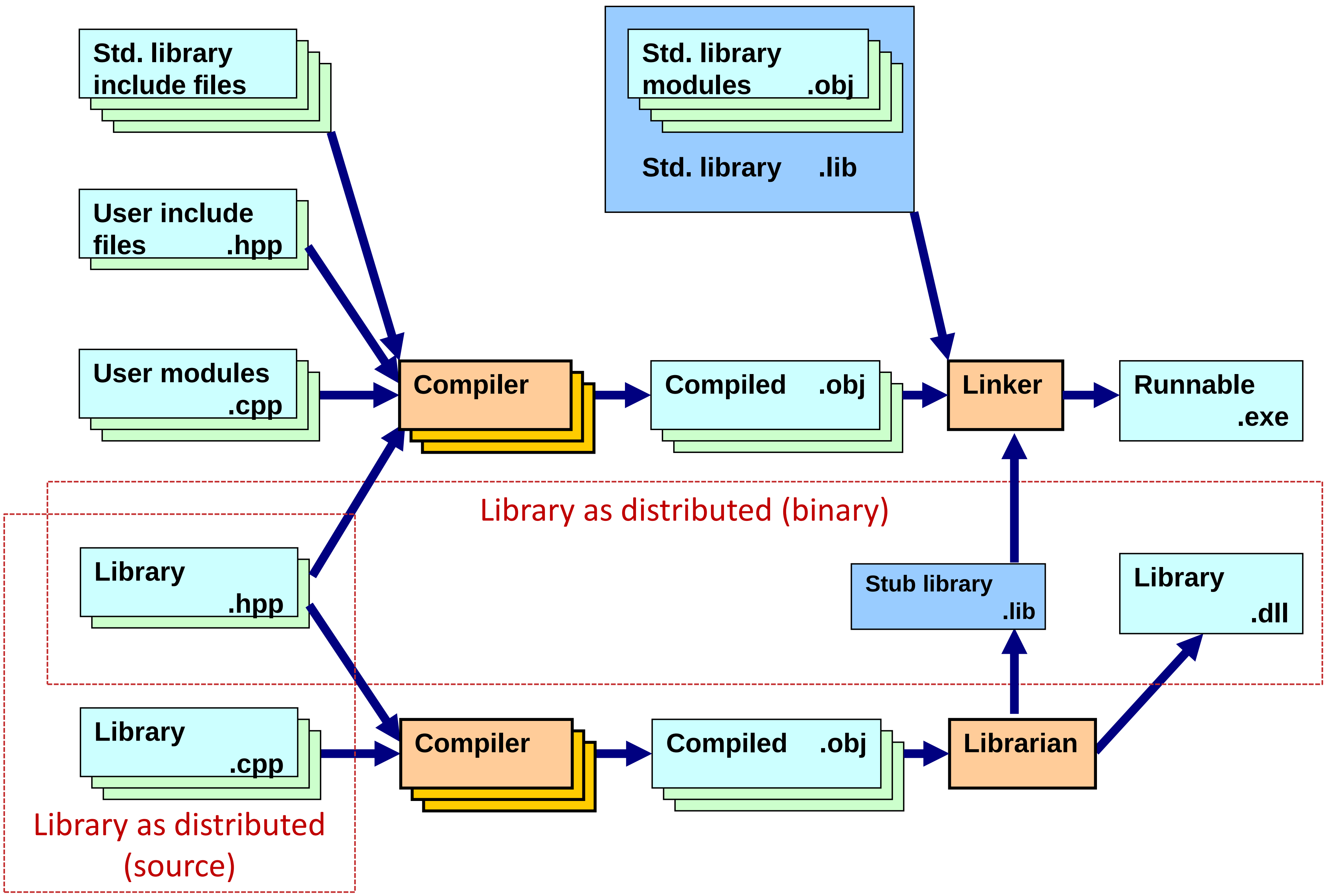
Integrated environment



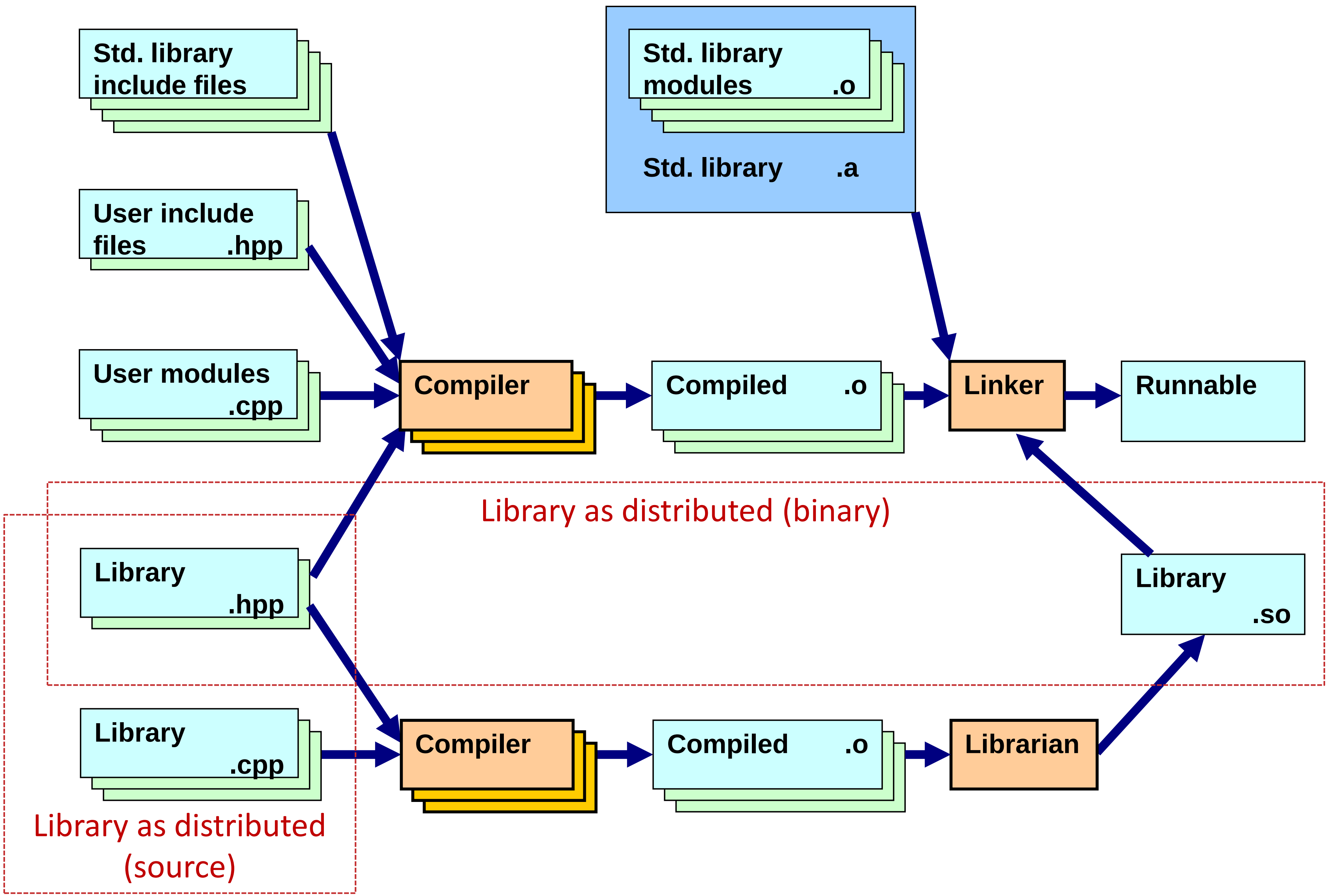
Static libraries



Dynamic libraries (Microsoft)



Dynamic libraries (Linux)



.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.

Declarations and definitions

Declarations and definitions

- Declaration
  - A construct to declare the existence (of a class/variable/function/...)
    - Identifier
    - Some basic properties
    - Ensures that (some) references to the identifier may be compiled
      - Some references may require definition
- Definition
  - A construct to completely define (a class/variable/function/...)
    - Class contents, variable initialization, function implementation
    - Ensures that the compiler may generate runtime representation
  - Every definition is a declaration
- Declarations allow (limited) use of identifiers without definition
    - Independent compilation of modules
    - Solving cyclic dependences
    - Minimizing the amount of code that requires (re-)compilation

Declarations and definitions

- One-definition rule #1:
  - One *translation unit*...
    - (*module*, i.e. one .cpp file and the .hpp files included from it)
  - ... may contain at most one definition of any item

- One-definition rule #2:
  - Program...
    - (i.e. the .exe file including the linked .dll files)
  - ... may contain at most one definition of a variable or a non-inline function
    - Definitions of classes, types or inline functions may be contained more than once (due to inclusion of the same .hpp file in different modules)
      - If these definitions are not identical, undefined behavior will occur
      - Beware of version mismatch between headers and libraries
    - Diagnostics is usually poor (by linker)

Exception handling

Exception handling

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance

```
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

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance
    - The value may be ignored

```
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

- Exceptions are "jumps"
  - Start: ***throw*** statement
  - Destination: ***try-catch*** block
    - Determined in run-time
  - The jump may exit a procedure
    - Local variables will be properly destructed by destructors
  - Besides jumping, a value is passed
    - The type of the value determines the destination
    - Typically, special-purpose classes
    - Catch-block matching can understand inheritance
    - The value may be ignored
    - There is an universal catch block

```
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

- Evaluating the expression in the throw statement
  - The value is stored "somewhere"
- Stack-unwinding
  - Blocks and functions are being exited
  - Local and temporary variables are destructed by calling destructors (user code!)
  - Stack-unwinding stops in the try-block whose catch-block matches the throw expression type
- catch-block execution
  - The throw value is still stored
    - may be accessed via the catch-block argument (typically, by reference)
  - "throw;" statement, if present, continues stack-unwinding
- Exception handling ends when the accepting catch-block is exited normally
  - Also using return, break, continue, goto
  - Or by invoking another exception

Exception handling

- Materialized exceptions
  - std::exception_ptr is a smart-pointer to an exception object
    - Uses reference-counting to deallocate
  - std::current_exception()
    - Returns (the pointer to) the exception being currently handled
    - The exception handling may then be ended by exiting the catch-block
  - std::rethrow_exception(p)
    - (Re-)Executes the stored exception
    - like a *throw* statement
  - This mechanism allows:
    - Propagating the exception to a different thread
    - Signalling exceptions in the promise/future mechanism

C++11

```
std::exception_ptr p;

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

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

Exception handling

- Throwing and handling exceptions is slower than normal execution
  - Compilers favor normal execution at the expense of exception-handling complexity
- Use exceptions only for rare events
  - Out-of-memory, network errors, end-of-file, ...

- Mark procedures which cannot throw by ***noexcept***

```
void f() noexcept
{ /*...*/
}
```

  - it may make code calling them easier (for you and for the compiler)
  - ***noexcept*** may be conditional

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

Exception handling

Standard exceptions

- <stdexcept>
- All standard exceptions are derived from class ***exception***
  - the member function ***what()*** returns the error message
- **bad_alloc**: not-enough memory
- **bad_cast**: dynamic_cast on references
- Derived from logic_error:
  - **domain_error, invalid_argument, length_error, out_of_range**
  - e.g., thrown by vector::at
- Derived from runtime_error:
  - **range_error, overflow_error, underflow_error**

- Hard errors (invalid memory access, division by zero, ...) are NOT signalized as exceptions
  - These errors might occur almost anywhere
  - The need to correctly recover via exception handling would prohibit many code optimizations

Exception-safe programming

Exception-safe programming

- Using throw a catch is simple
- Producing code that works correctly in the presence of exceptions is hard
  - Exception-safety
  - Exception-safe programming

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

- If new int[200] throws, the int[100] block becomes inaccessible
- If g() throws, two blocks become inaccessible

Exception-safe programming

- The use of smart pointers solves some problems related to exception safety

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

- RAII: Resource Acquisition Is Initialization
  - Constructor allocates resources
  - Destructor frees the resources
    - Even in the case of an exception

```
std::mutex my_mutex;

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

Exception-safe programming

- Using throw a catch is simple
- Producing code that works correctly in the presence of exceptions is hard
  - 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;
}
```

Exception-safe programming

- Language-enforced rules
  - Destructors may not end by throwing an exception
  - Constructors of static variables may not end by throwing an exception
  - Copying of exception objects may not throw
- Compilers sometimes generate implicit try blocks
  - When constructing a compound object, a part constructor may throw
    - Array allocation
    - Class constructors
  - The implicit try block destructs previously constructed parts and rethrows

Exception-safe programming

- Theory

- (Weak) exception safety
  - Exceptions does not cause *inconsistent* state
    - No memory leaks
    - No invalid pointers
    - Application invariants hold
    - ...?

- Strong exception safety
  - Exiting function by throwing means *no change* in (observable) state
  - *Observable state* = public interface behavior
  - Also called "Commit-or-rollback semantics"

Class patterns

copy/move

- Most-frequent cases
  - A harmless class
    - No copy/move method, no destructor
    - No dangerous data members (raw pointers)
  - A class containing dangerous members
    - Compiler-generated behavior (default) would not work properly
    - No move support (before C++11, still functional but not optimal)

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

    - Full copy/move support

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

- Handling data members in constructors and destructors
  - Numeric types
    - Explicit initialization recommended, no destruction required
    - Compiler-generated copy/move works properly
  - Structs/classes
    - If they have no copy/move methods, they behave as if their members were present directly
    - If they have copy/move methods, they usually do not require special handling
      - Special handling required if the outer class semantics differ from the inner class (e.g., using smart pointers to implement containers)
  - Containers and strings
    - Behave as if their members were present directly
      - Containers are initialized as empty - no need to initialize even containers of numeric types

copy/move

- Data members - links without ownership
  - References (U&)
      - for class members, use of pointers U* is preferred over U&
    - Explicit initialization required, destruction not required
    - Copy/move constructors work smoothly
    - Copy/move operator= is impossible
      - if assignment is needed, use std::ref_wrapper< T> instead of T &
  - Raw pointers (U*) without ownership semantics
      - Proper deallocation is ensured by someone else
    - Explicit initialization required, destruction not required
    - Copy/move work smoothly

- Data members - smart pointers
  - std::unique_ptr<U>
    - Explicit initialization not required (nullptr by default)
    - Explicit destruction not required (smart pointers deallocate automatically)
    - Copying is impossible
      - If copying is required, it must be implemented by duplicating the linked object
    - Move methods work smoothly
  - std::shared_ptr<U>
    - Explicit initialization not required (nullptr by default)
    - Explicit destruction not required (smart pointers deallocate automatically)
    - Copying works as sharing
      - If sharing semantics is not desired, other methods must be adjusted
        - all modifying operations must ensure a private copy of the linked object
    - Move methods work smoothly

Class patterns

- POD: Plain-Old-Data
  - Public data members
  - The user is responsible for initialization

```
class T {
public:
  std::string x_;
};
```

- struct often used instead of class

```
struct T {
  std::string x_;
};
```

Class patterns

- All data-members harmless
  - Every data member have its own constructor
  - The class does not require any constructor

```
class T {
public:
  // ...
  const std::string & get_x() const { return x_; }
  void set_x( const std::string & s) { x_ = s; }
private:
  std::string x_;
};
```

Class patterns

- All data-members harmless
  - Every data member have its own constructor
  - Constructor enables friendly initialization
    - Due to language rules, the parameter-less constructor is often needed too

```
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)
  {}
  // ... other methods ...
private:
  std::string x_, y_;
};
```

Class patterns

- Some slightly dangerous elements
  - Some elements lack suitable default constructors
    - Numeric types (including bool, char), observer pointers (T *, const T *)
  - A constructor is required to properly initialize these elements
    - Consequently, default (parameterless) constructor is (typically) also required
    - One-parameter constructors marked explicit

```
class T {
public:
  T() : x_( 0), y_( 0) {}
  explicit T( int s) : x_( s), y_( 0) {}
  T( int s, int t)
    : x_( s), y_( t)
  {}
  // ... other methods ...
private:
  int x_, y_;
};
```

copy/move

- Less frequent cases
  - A non-copyable and non-movable class
    - E.g., dynamically allocated "live" objects in simulations

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

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

    - The delete keyword prohibits automatic default for copy methods
    - Language rules prohibit automatic default for move methods
    - A destructor may be required

  - A movable non-copyable class
    - E.g., an owner of another object (like std::unique_ptr< U>)

```
T( T && x);

T & operator=( T && x);

~T();
```

    - Language rules prohibit automatic default for copy methods
    - A destructor is typically required

Class patterns

- Classes containing unique_ptr
  - Uncopiable class
    - But moveable

```
class T {
public:
  T() : p_( std::make_unique< Data>()) {}
private:
  std::unique_ptr< Data> p_;
};
```

Class patterns

- Classes containing unique_ptr
  - Copying enabled

```
class T {
public:
  T() : p_(std::make_unique< Data>()) {}
  T( const T & x) : p_(std::make_unique< 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_;
};
```

Class patterns

- Abstract class
  - Copying/moving prohibited

```
class T {
protected:
  T() {}
  T( const T & x) = delete;
  T & operator=( const T & x) = delete;
public:
  virtual ~T() {}  // required for proper deletion of objects
};
```

Class patterns

- Abstract class
  - Cloning support

```
class T {
protected:
  T() {}
  T( const T & x) = default;         // descendants will need it to implement clone
  T & operator=( const T & x) = delete;
public:
  virtual ~T() {}
  virtual std::unique_ptr< T> clone() const = 0;
};
```

- Data members - links with ownership
  - Raw pointers (U*) with unique ownership
    - Our class must deallocate the remote object properly
    - Explicit initialization required (allocate or set to zero)
    - Destruction is required (deallocate if not zero)
    - Copy methods must allocate new space a copy data
    - Move methods must clear links in the source object
    - In addition, copy/move operator= must clean the previous contents
  - Raw pointer (U*) with shared ownership
    - Our class must count references and deallocate if needed
    - Explicit initialization required (allocate or set to zero)
    - Destruction is required (decrement counter, deallocate if needed)
    - Copy methods must increment counter
    - Move methods must clear links in the source object
    - In addition, copy/move operator= must clean the previous contents

Class patterns

- Some very dangerous elements [avoid whenever possible]
  - Raw pointers with (exclusive/shared) ownership semantics
  - copy/move constructor/operator= and destructor required
    - Some additional constructor (e.g. default) is also required

```
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_;
};
```