Programming in C++

Course credits

- Conditions
  - Exams
    - abc-tests
    - January to February - registration in SIS
  - Passing practical programming tests
    - in lab, approx. 3 hours, common sessions for all groups - registration in SIS
    - 1st attempts - 2nd half of January
    - 2nd attempts - 1st half of February
    - 3rd attempts - April
  - Creating an individual project
    - Agreement on project assignment - until end of November
    - Beta version until March 31, 2016
    - Final version including documentation until May 20, 2016
  - Reasonable participation in labs
  - Homework assignments
- Conditions may be individually adjusted:
  contact your lab teacher during October
  - Erasmus students may need dates and deadlines sooner

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!

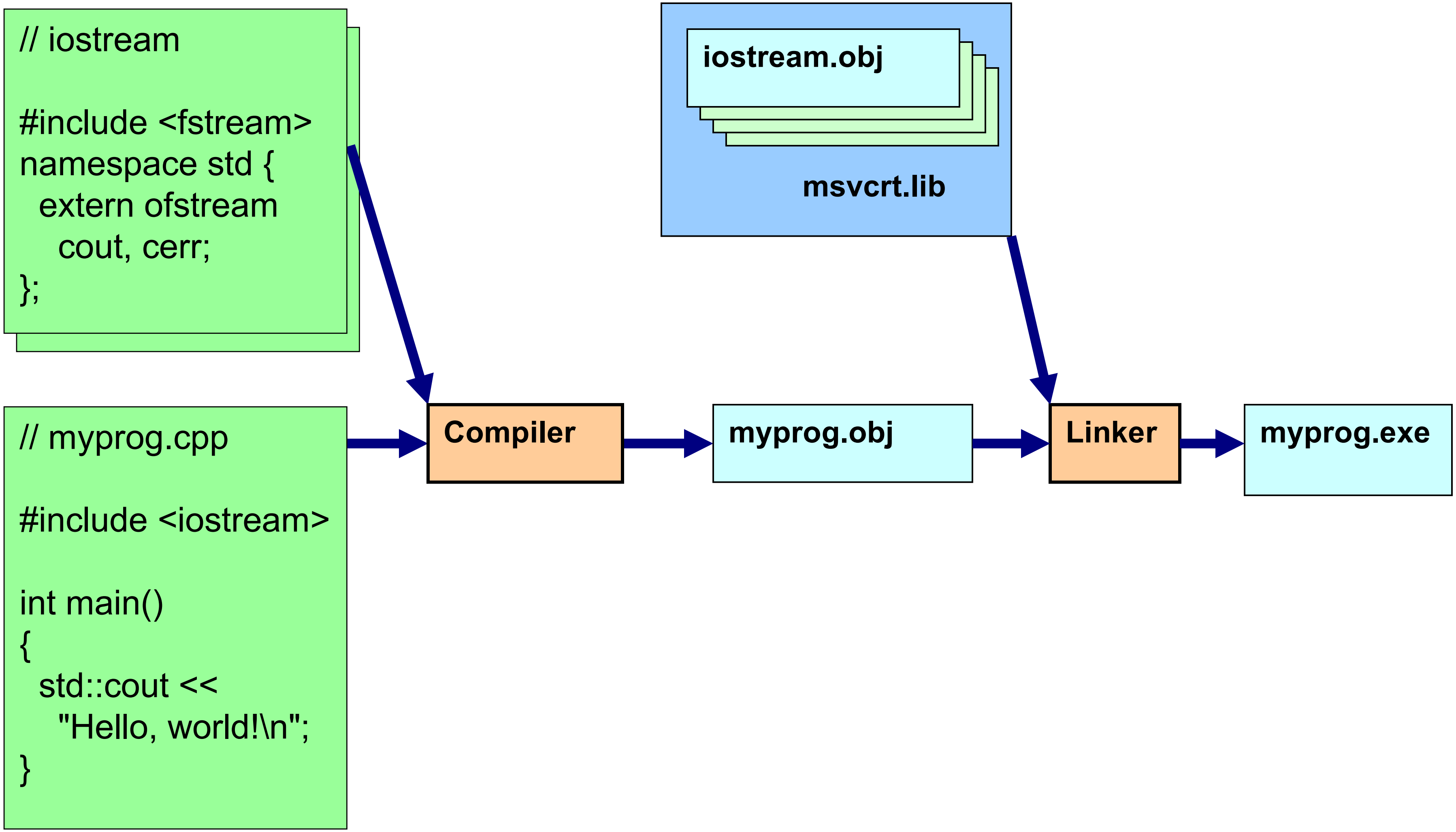
```
// main.cpp
#include "world.hpp"

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

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

#include <vector>
#include <string>

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

#endif
```

```
// 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

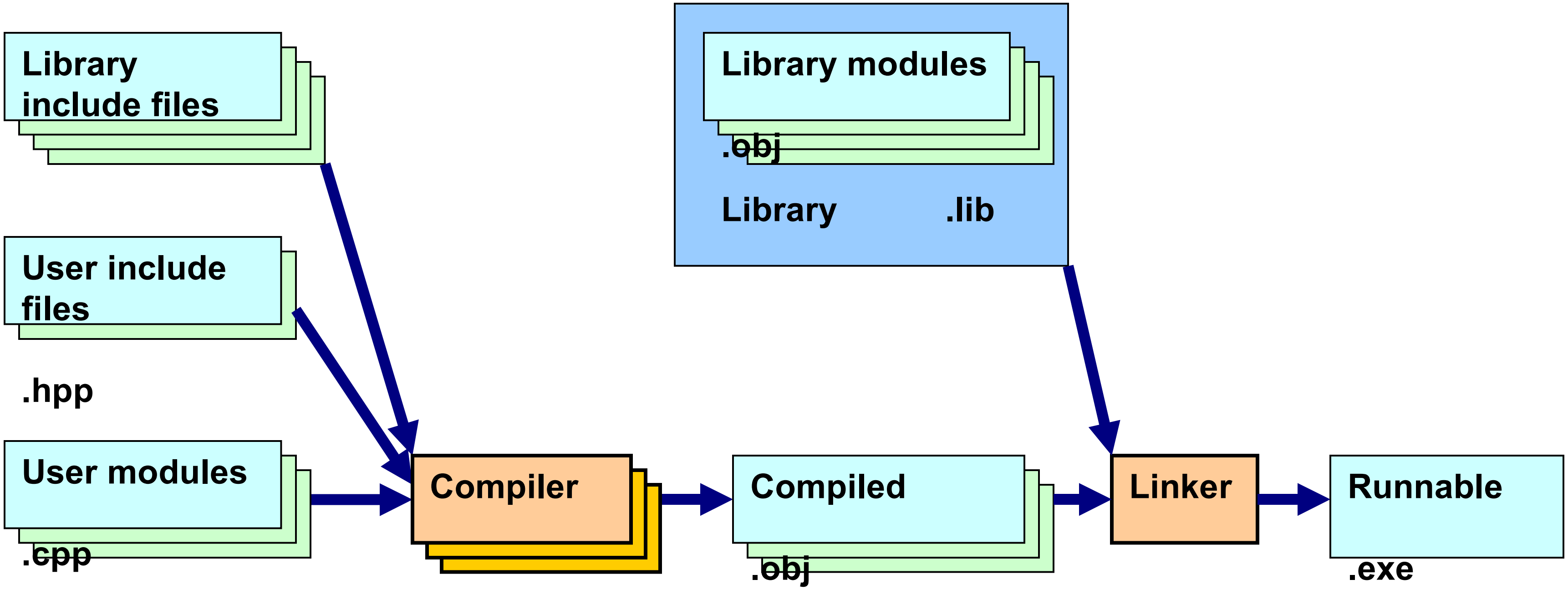
Single-module programs - static linking



```
// iostream

#include <fstream>
namespace std {
  extern ofstream
    cout, cerr;
};
```

```
// myprog.cpp

#include <iostream>

int main()
{
  std::cout <<
    "Hello, world!\n";
}
```

Multiple-module programs



Library include files

User include files

.hpp

User modules

.cpp

Compiler

Library modules

.obj

Library .lib

Compiled

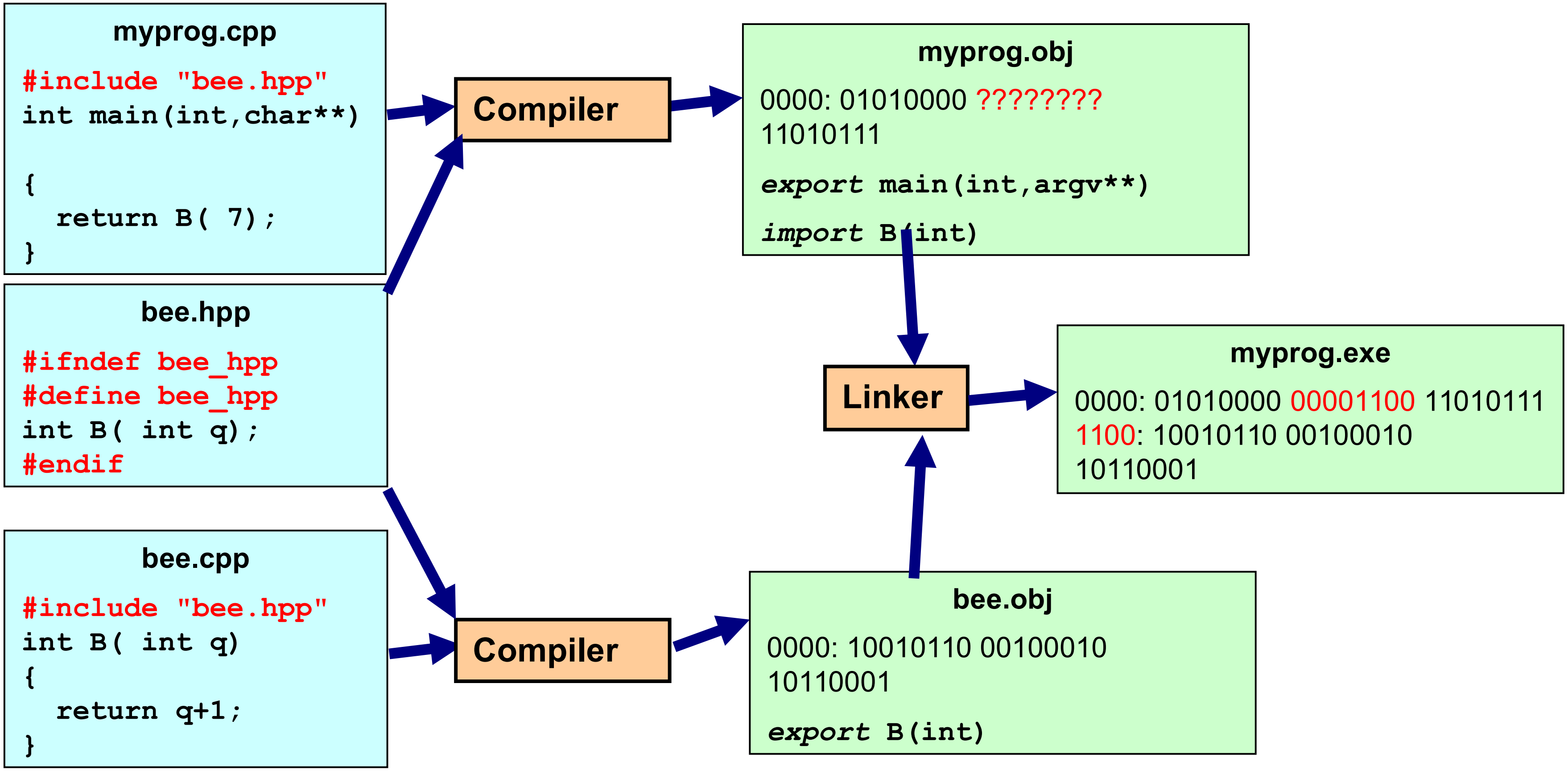
.obj

Linker

Runnable

.exe

Module interfaces and linking



myprog.cpp

```
#include "bee.hpp"
int main(int,char**)

{
  return B( 7);
}
```

bee.hpp

```
#ifndef bee_hpp
#define bee_hpp
int B( int q);
#endif
```

bee.cpp

```
#include "bee.hpp"
int B( int q)
{
  return q+1;
}
```

Compiler

Compiler

myprog.obj

0000: 01010000 ????????
11010111

export `main(int,argv**)`

import `B(int)`

bee.obj

0000: 10010110 00100010
10110001

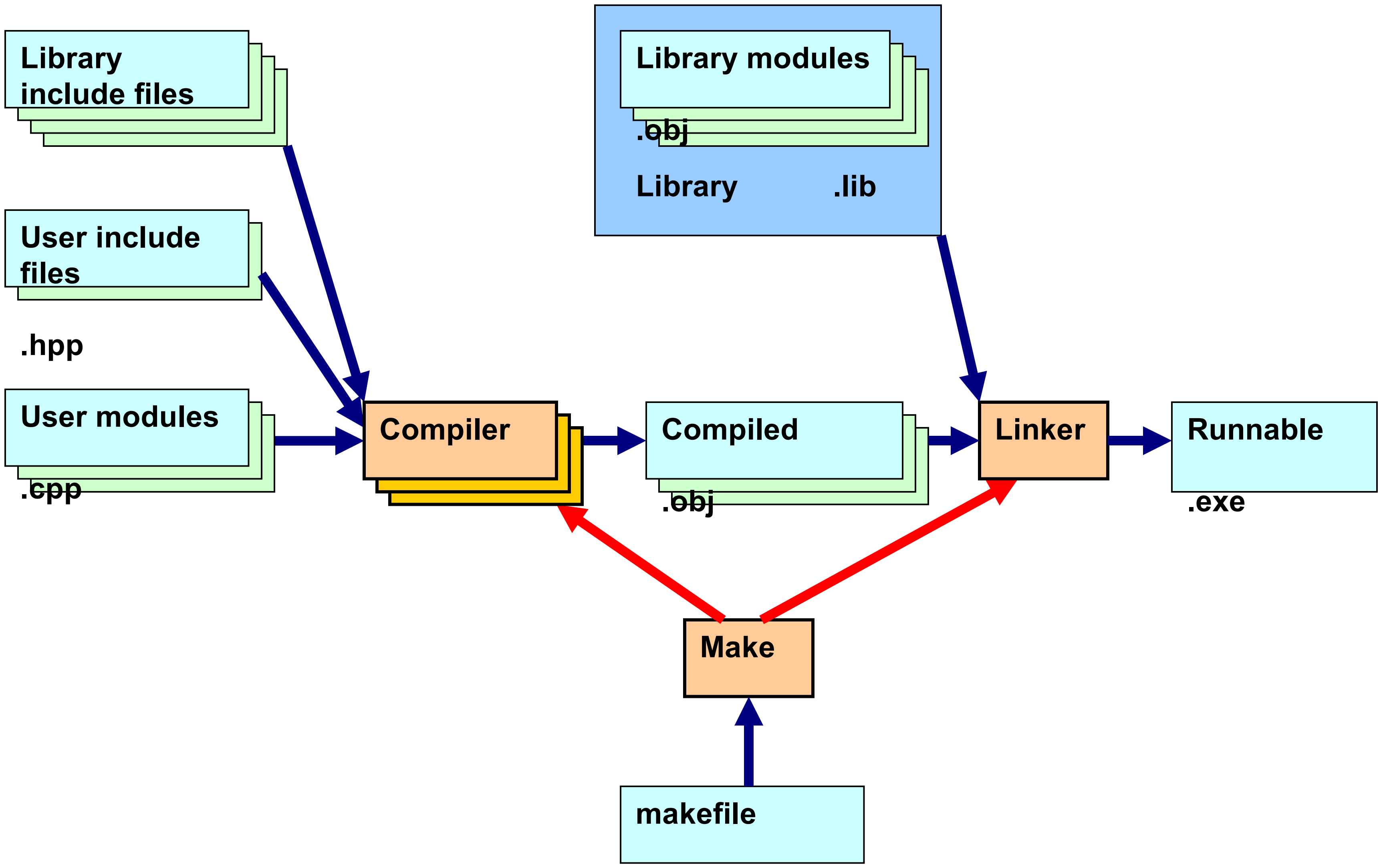
export `B(int)`

Linker

myprog.exe

0000: 01010000 00001100 11010111
1100: 10010110 00100010
10110001

make



Library include files

User include files

.hpp

User modules

.cpp

Library modules

.obj

Library .lib

Compiler

Compiled

.obj

Linker

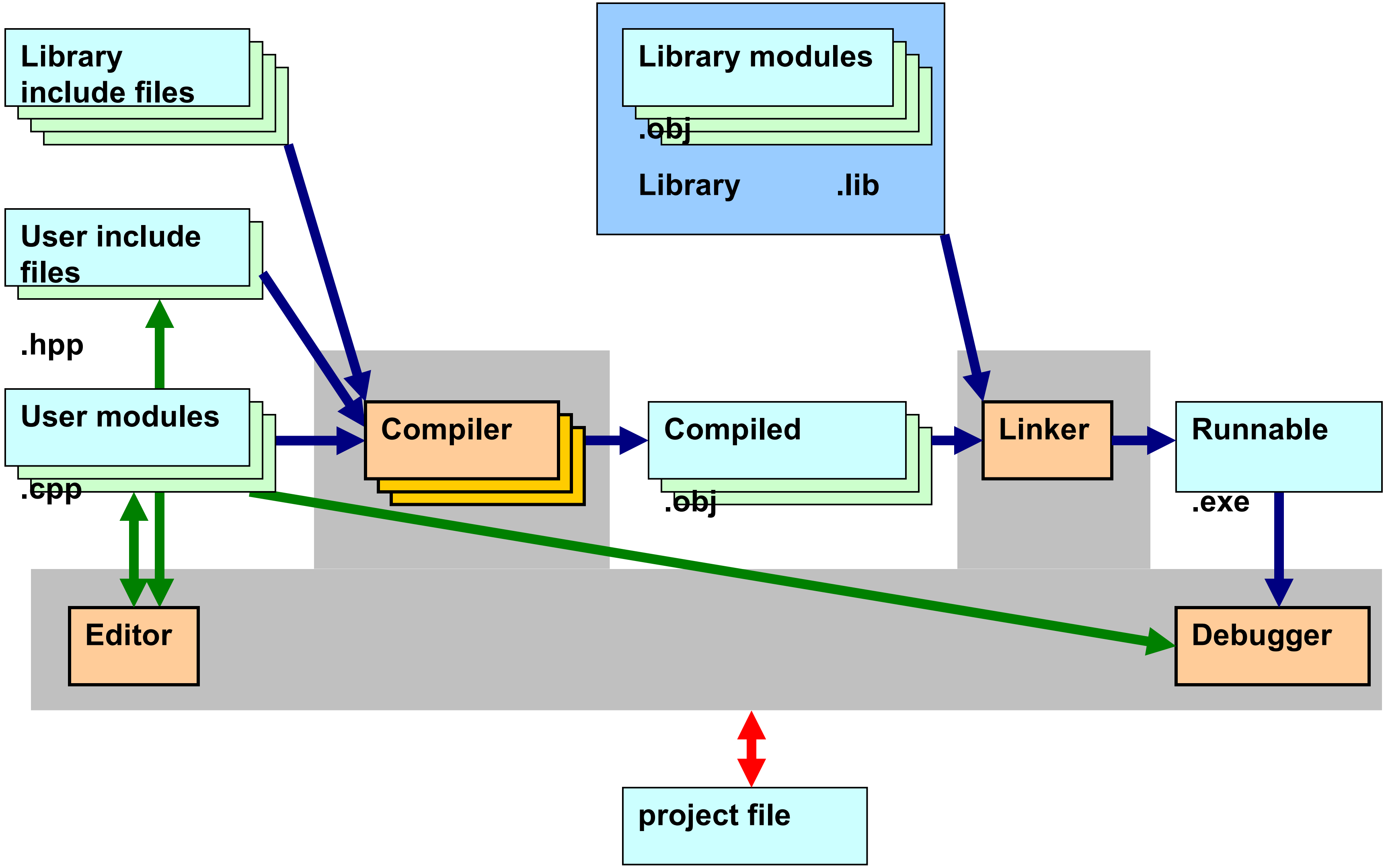
Runnable

.exe

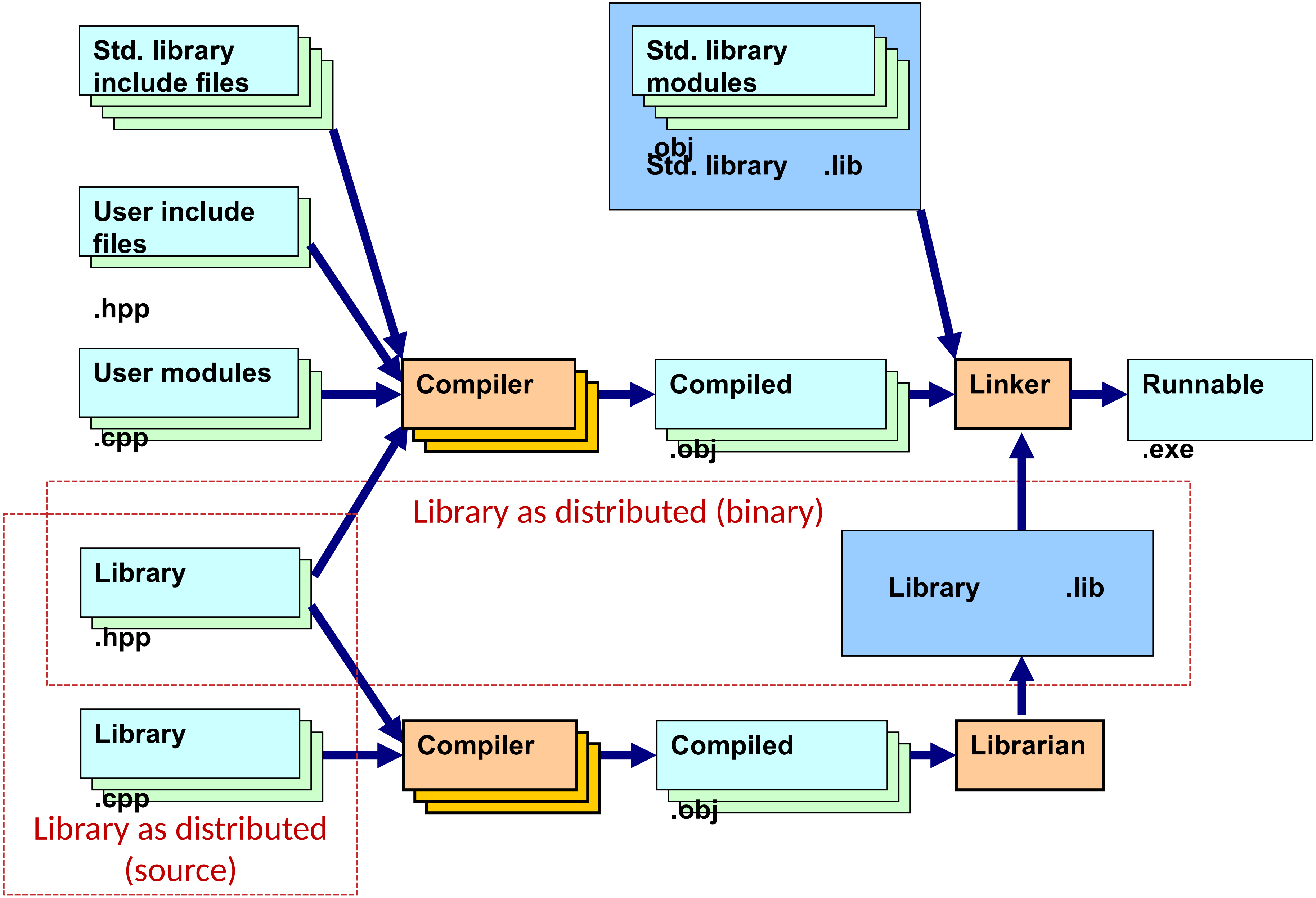
Make

makefile

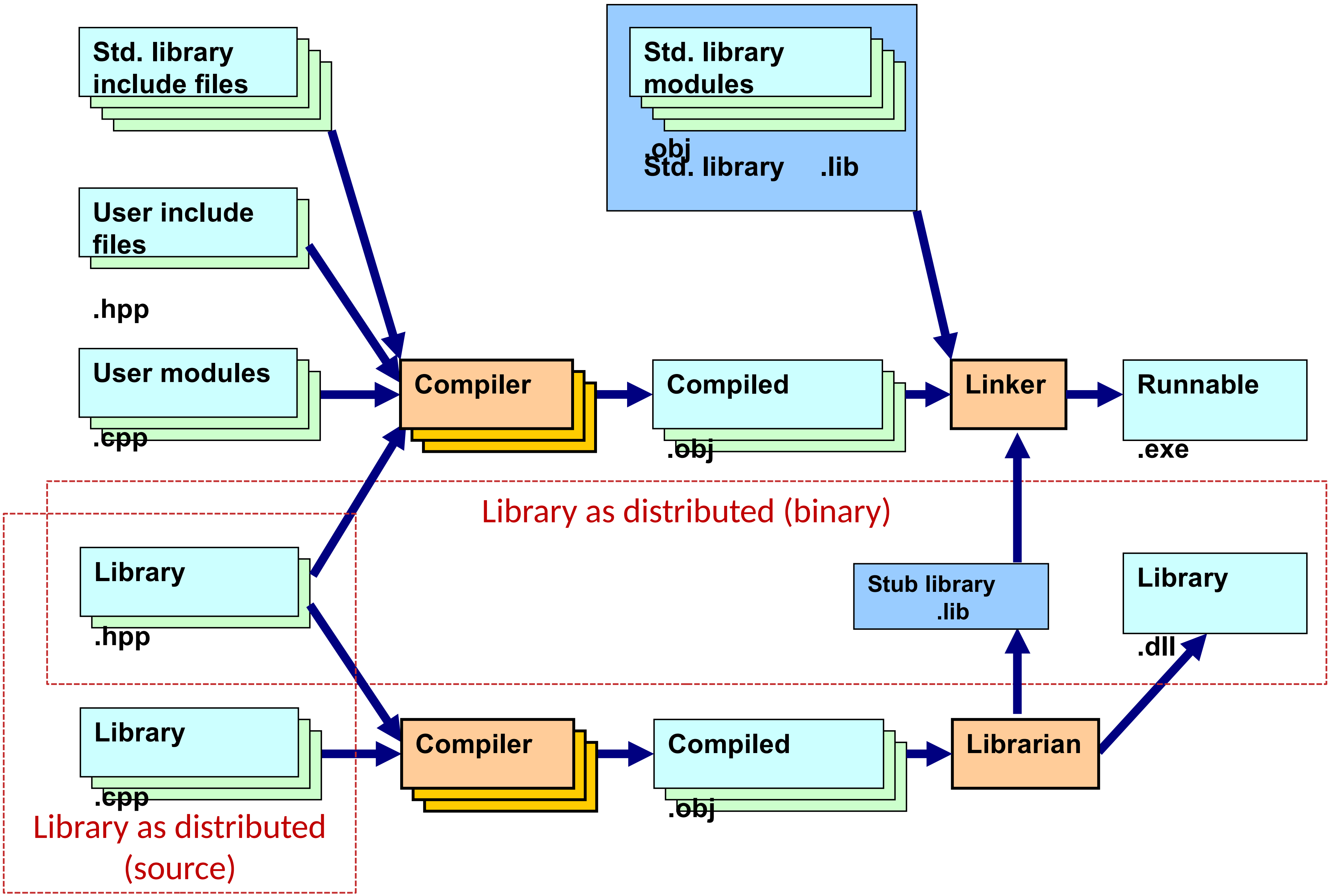
Integrated environment



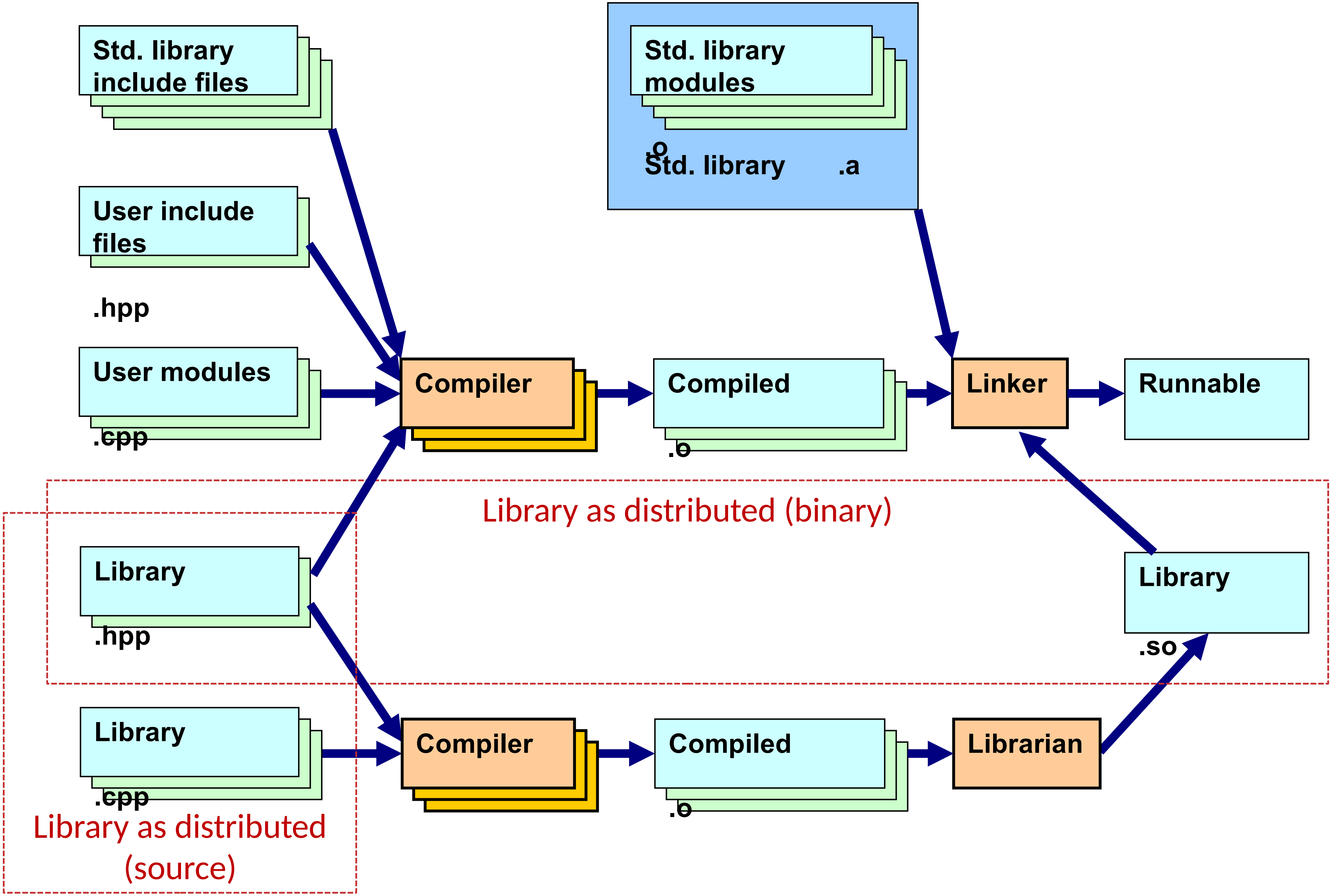
Static libraries



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)

Class and type definitions

<table>
<tr><th></th><th>Declaration</th><th>Definition</th></tr>
<tr><td>Class</td><td><pre>class A;</pre></td><td><pre>class A {
  ...
};</pre></td></tr>
<tr><td>Structure (almost equivalent to class)</td><td><pre>struct A;</pre></td><td><pre>struct A {
  ...
};</pre></td></tr>
<tr><td>Union (unusable in C++)</td><td><pre>union A;</pre></td><td><pre>union A {
  ...
};</pre></td></tr>
<tr><td>Named type</td><td></td><td><pre>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;</pre></td></tr>
<tr><td>C++11 style of named types</td><td></td><td><pre>using A2 = A;
using AFP2 = A (*)();</pre></td></tr>
</table>

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

Global variable

Declaration

```
extern int x, y, z;
```

Definition

```
int x; int y = 729; int z(729);
int u{729};
```

C++11

Static member variable

Declaration

```
class A {
  static int x, y, z;
};
```

Definition

```
int A::x; int A::y = 729;
int A::z( 729);
int A::z{ 729};
```

C++11

Constant member

Definition

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

Static local variable

Definition

```
void f() {
  static int x;
  static int y = 7, z( 7);
  static int u{ 7};
}
```

C++11

Nonstatic member variable

Definition

```
class A {
  int x, y;
};
```

Nonstatic local variable

Definition

```
void f() {
  int x;
  int y = 7, z( 7);
  int u{ 7};
};
```

C++11

Storage classes

- Where data reside...
  - Static storage
    - Global, static member, static local variables, string constants
      - One instance per process
    - Allocated by compiler/linker/loader (listed in .obj/.dll/.exe)
  - 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 function invocation (execution of defining statement)
    - Placement by compiler, space allocated by compiler-generated instructions
  - Dynamic allocation
    - new/delete operators
      - The programmer is responsible for deallocation, no garbage collection
    - Allocation by library routines
      - Significantly slower than other storage classes

Storage classes

- Where data reside...
  - Static storage

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

  - Thread-local storage

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

  - Automatic storage (stack or register)

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

  - Dynamic allocation

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

Dynamic allocation

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

```
#include <memory>
```

  - one owner (pointer cannot be copied)
    - no runtime cost (compared to T *)

```
void f() {
  std::unique_ptr< T> p = new 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 to q
}
```

- 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

Dynamic allocation

- Dynamic allocation is slow
  - compared to static/automatic storage
  - the reason is cache behavior, not the allocation itself
- Use dynamic allocation only when necessary
  - variable-sized or large arrays
  - polymorphic containers (objects with inheritance)
  - object lifetimes not corresponding to function invocations
- Avoid data structures with individually allocated items
  - linked lists, binary trees, ...
    - std::list, std::map, ...
  - prefer B-trees (yes, also in memory) or hash tables
  - 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

Arrays

	Homogeneous	**Polymorphic**

Fixed size	`static const std::size_t n = 3;`
	`std::array< T, n> a;`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`std::tuple< T1, T2, T3> a;`
		
		`std::get< 0>( a) = /*...*/;`
		`std::get< 1>( a).f();`

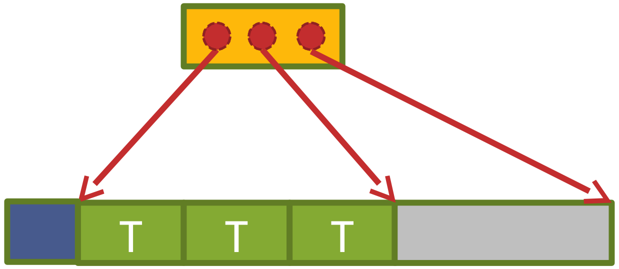
Variable size	`std::size_t n = /*...*/;`
	`std::vector< T> a(n);`
	
	`a[ 0] = /*...*/;`
	`a[ 1].f();`
		`std::vector< std::unique_ptr< Tbase>> a;`
		`a.push_back( new T1);`
		`a.push_back( new T2);`
		`a.push_back( new T3);`
		
		`a[ 1]->f();`

Array layouts

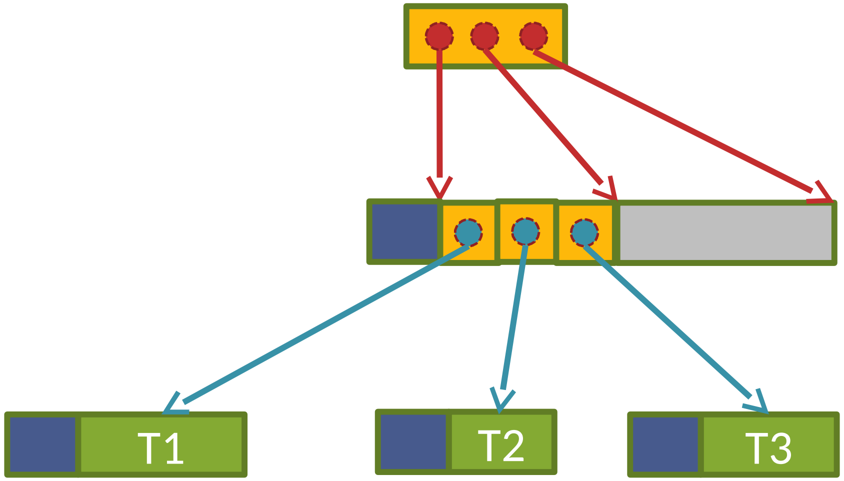
`std::array< T, 3>`

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

`std::vector< T>`



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



Frequently used data types

Selected number types

`bool`	false, true
`char`	character (ASCII, 8 bit)
`std::wchar_t`	character (Unicode, 16/32 bit)
`int`	signed integer (~32 bit)
`unsigned`	unsigned integer (~32 bit)
`long long`	extra large signed integer (~64 bit)
`unsigned long long`	extra large unsigned integer (~64 bit)
`std::size_t`	unsigned integer large enough for array sizes (32/64 bit)
`double`	"double precision" floating-point number (Intel: 64 bit)
`long double`	extended precision floating-point number (Intel: 80 bit)
`std::complex<double>`	complex number of double precision

Important non-number types

`std::string`	string (containing char)
`std::wstring`	string (containing std::wchar_t)
`std::istream`	input stream (containing char)
`std::wistream`	input stream (containing std::wchar_t)
`std::ostream`	output stream (containing char)
`std::wostream`	output stream (containing std::wchar_t)
`struct T { … }`	structure (almost equivalent to class)
`std::pair<T1,T2>`	pair of T1 and T2
`std::tuple<T1,...>`	k-tuple of various types
`std::array<T,n>`	fixed-size array of T
`std::vector<T>`	variable-size array of T
`std::list<T>`	doubly linked list of T
`std::map<K,T>`	ordered associative container of T indexed by K
`std::multimap<K,T>`	ordered associative container with multiplicity of keys
`std::unordered_map<K,T>`	hash table of T indexed by K
`std::unordered_multimap<K,T>`	hash table with multiplicity of keys

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 const 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:
  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;
  static const 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 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 const t c = 1;
  static t f( t p)
  { return p + v; }
  static t v;  // declaration of X::v
};
```

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

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

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

- A class may become a template argument

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

Namespace

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

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

- Namespace may be reopened
  - Namespace may be split into several header files
  - Definitions of namespace members must reopen it

```
namespace X {
  t v = c;     // definition of X::v
};
```

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

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

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

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

  - Element of a larger type

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

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

Class with data members

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

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

```
Y v1;
std::unique_ptr< Y> p = new Y;
```

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

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

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

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

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

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

Pointer vs. value

Forms of pointers in C++

- References

`T &`

`const T &`

  - Built in C++
  - 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
  - Manual allocation/deallocation

- 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
  - Operators to access the referenced value same as with raw pointers (*p, p->a)
  - Pointer arithmetics allows to access adjacent values in the container

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

Raw pointers (C++)

```
class T {
public:
  int a;
};

void f( T * z)
{
  z->a = 3;
}

void g()
{
  T * x = new T;
    // allocation

  x->a = 1;

  T * y = x;
    // second pointer

  y->a = 2;
    // x->a == 2

  f( x);
    // x->a == 3

  delete x;
    // manual deallocation
}
```

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

Smart pointers (C++)

```
class T {
public:
  int a;
};

void f( T * z)
{
  z->a = 3;
}

void g()
{
  std::shared_ptr< T> x =
    std::make_shared< T>();
    // allocation

  x->a = 1;

  std::shared_ptr< T> y = x;
    // second pointer

  y->a = 2;
    // x->a == 2

  f( x);
    // x->a == 3

  // automatic deallocation
  // when pointers are destructed
}
```

C#/Java vs. C++

Reference types (C#,Java)

```
class T {
  public int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x = new T();
      // allocation

    x.a = 1;

    T y = x;
      // second reference

    y.a = 2;
      // x.a == 2

    f( x);
      // x.a == 3

    // garbage collector will later
    // reclaim the memory when needed
  }
}
```

References (C++)

```
class T {
public:
  int a;
};

void f( T & z)
{
  z.a = 3;
}

void g()
{
  T x;    // automatic storage (stack)

  x.a = 1;

  T & y = x;
    // a reference to the stack object

  y.a = 2;
    // x.a == 2

  f( x);
    // x.a == 3

  // x is destructed on exit
}
```

C#/Java vs. C++

Value types (C#)

```
struct T {
  int a;
}

class test {
  static void f( T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x;
      // creation

    x.a = 1;

    T y = x;
      // a copy

    y.a = 2;
      // x.a == 1

    f( x);
      // x.a == 1

      // destruction on exit
  }
}
```

Values (C++)

```
class T {
public:
  int a;
};

void f( T z)
{
  z.a = 3;
}

void g()
{
  T x;
    // creation

  x.a = 1;

  T y = x;
    // a copy

  y.a = 2;
    // x.a == 1

  f( x);
    // x.a == 1

    // destruction on exit
}
```

C#/Java vs. C++

Passing value types by reference (C#)

```
struct T {
  int a;
}

class test {
  static void f( ref T z)
  {
    z.a = 3;
  }

  static void g()
  {
    T x;
      // creation

    x.a = 1;

    f( ref x);
      // x.a == 3
  }
}
```

Passing by lvalue reference (C++)

```
class T {
public:
  int a;
};

void f( T & z)
{
  z.a = 3;
}

void g()
{
  T x;

  x.a = 1;

  f( x);
    // x.a == 3
}
```

C#/Java vs. C++

Passing reference types by reference (C#)

```
class T {
  public int a;
}

class test {
  static void f( ref T z)
  {
    z = new T();
      // allocation of another object
  }

  static void g()
  {
    T x = new T();
      // allocation

    f( ref x);
      // x is now a different object

    // deallocation later by GC
  }
}
```

Passing smart pointers by reference (C++)

```
class T {
public:
  int a;
};

void f( std::unique_ptr<T> & z)
{
  z = new T;
    // allocation of another object
    // deallocation of the old object
}

void g()
{
  std::unique_ptr< T> x = new T;
    // allocation

  f( x);
    // *x is now a different object

  // deallocation by destruction of x
}
```

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

- 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
`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)
`const T &`	Read the contents	During a call/statement	Nothing (usually)

Transferring unique ownership

```
channel ch;

void send_hello()
{
  std::unique_ptr< packet> p = new 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 { /*...*/ };

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()
{
  std::unique_ptr< packet> p = new 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 { /*...*/ };


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


  std::unique_ptr< packet> receive()
  {
    std::unique_ptr< packet> 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()
{
  std::shared_ptr< channel> ch =
    std::make_shared< channel>();
  s.reset( new sender( ch));
  r.reset( new 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.reset( new channel);
  s.reset( new sender( ch.get()));
  r.reset( new 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.reset( new sender( ch));
  r.reset( new 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.reset( new channel);
  sender s( ch.get());
  recipient r( ch.get());
  do_it( s, r);
  ch.reset( new channel);
  do_it( s, r);
}
```

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

Allowing access temporarily

```
channel ch;

void send_hello()
{
  std::unique_ptr< packet> p = new 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

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
}
```

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();
```

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

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

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

STL

- 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

STL

- 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

```
  my_iterator i1 = begin( c1); // also c1.begin()
  my_const_iterator i2 = cbegin( c1);   // also c1.cbegin(), begin( c1), c1.begin()
  my_const_iterator 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 = /*...*/;
  my_iterator i4 = c1.find( k);
  my_const_iterator i5 = c2.find( k);
```

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

```
  for ( my_iterator 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

```
  my_container::difference_type delta = i4 - c1.begin();// number of objects left to i4
  my_iterator i7 = c1.end() - delta;    // the same distance from the opposite end
  if ( i4 < i7 )
    my_value 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 (my_iterator 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

```
std::unique_ptr< T> p( new 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

```
c4.emplace_back( new T);
c3.insert( 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);
```

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 (pomocí 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);
```

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 expressions

- Lambda expression

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

  - Declares a class

```
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
  - 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

Constructors and destructors

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

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

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();
```

copy/move

- Less frequent cases
  - A non-copiable 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-copiable 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

copy/move

- 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&)
    - Explicit initialization required, destruction not required
    - Copy/move constructors work smoothly
    - Copy/move operator= is impossible
  - Raw pointers (U*) without ownership semantics
      - Proper deallocation is ensured by someone else
    - Explicit initialization required, destruction not required
    - Copy/move work smoothly

copy/move

- 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

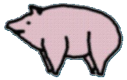
copy/move

- 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

Conversions

Special member functions

- Conversion constructors

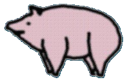


```
class T {
  T( U x);
};
```

  - 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 ) { /* ... */ }
```

Class patterns

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 parameterless 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)
  {}
  // ... metody ...
private:
  std::string x_, y_;
};
```

Class patterns

- Some slightly dangerous elements
  - Some elements lack suitable default constructors
    - Numeric types, including bool, char
  - 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)
  {}
  // ... metody ...
private:
  int x_, y_;
};
```

Class patterns

- Some very dangerous elements
    - 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_;
};
```

Class patterns

- Classes containing unique_ptr
  - Uncopiable class
    - But movable

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

Class patterns

- Classes containing unique_ptr
  - Copying enabled

```
class T {
public:
  T() : p_( new Data) {}
  T( const T & x) : p_( new Data( * x.p_)) {}
  T( T && x) = default;
  T & operator=( const T & x) { return operator=( T( x));}
  T & operator=( T && x) = default;
private:
  std::unique_ptr< Data> p_;
};
```

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;
};
```

Inheritance

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

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

- Derived class is a descendant of Base class
  - Contains all types, data elements and functions of Base
  - 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 void f() { /* ... */ }
};

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

Virtual functions

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

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

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

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

Inheritance and destructors

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

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

```
Base * p = new Derived;

delete p;
```

- If an object is destroyed using delete applied to a pointer to its base class, the destructor of the base class must be virtual
- Rule of thumb:
  - Every abstract class must have a virtual destructor
    - There is no additional cost (there are other virtual functions)
    - It will be probably needed

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
```

  - Abstract classes usually contain no data
  - Interfaces are 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 &

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]

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

Templates

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

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

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

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

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

- Type templates [C++11]

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

Templates

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

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

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

```
int a, b, c;

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

    - Explicitly

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

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

Templates

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

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

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

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

      - explicit specialization

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

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

Writing templates

- Compiler needs hints from the programmer
  - **Dependent names** have unknown meaning/contents
    - type names must be explicitly designated

```
template< typename T> class X

{

  typedef typename T::B U;

  typename U::D p;

  typename Y<T>::C q;

  void f() { T::D(); }          // T::D is not a type

}
```

- explicit template instantiations must be explicitly designated
- members inherited from dependent classes must be explicitly designated

```
template< typename T> class X : public T

{

  void f() { return this->a; }

}
```