SELA

Modern C++ (11 / 14 / 17)

by

Dr. Miri (Kopel) Ben-Nissan

Copyright © SELA Software & Education Labs, Ltd. | 14-18 Baruch Hirsch St., Bnei Brak 51202, Israel | www.selagroup.com

Modern C++ : Introduction

- Modern C++ is aimed to **make C++ programming easier**, **less error-prone**, and **more productive**.

- **No more traditional low-level language.**
- Instead - containers, smart pointers, or other RAII techniques, emphasizes exceptions to report errors, passing objects by value through move semantics, writing algorithms instead of loops, and so on.

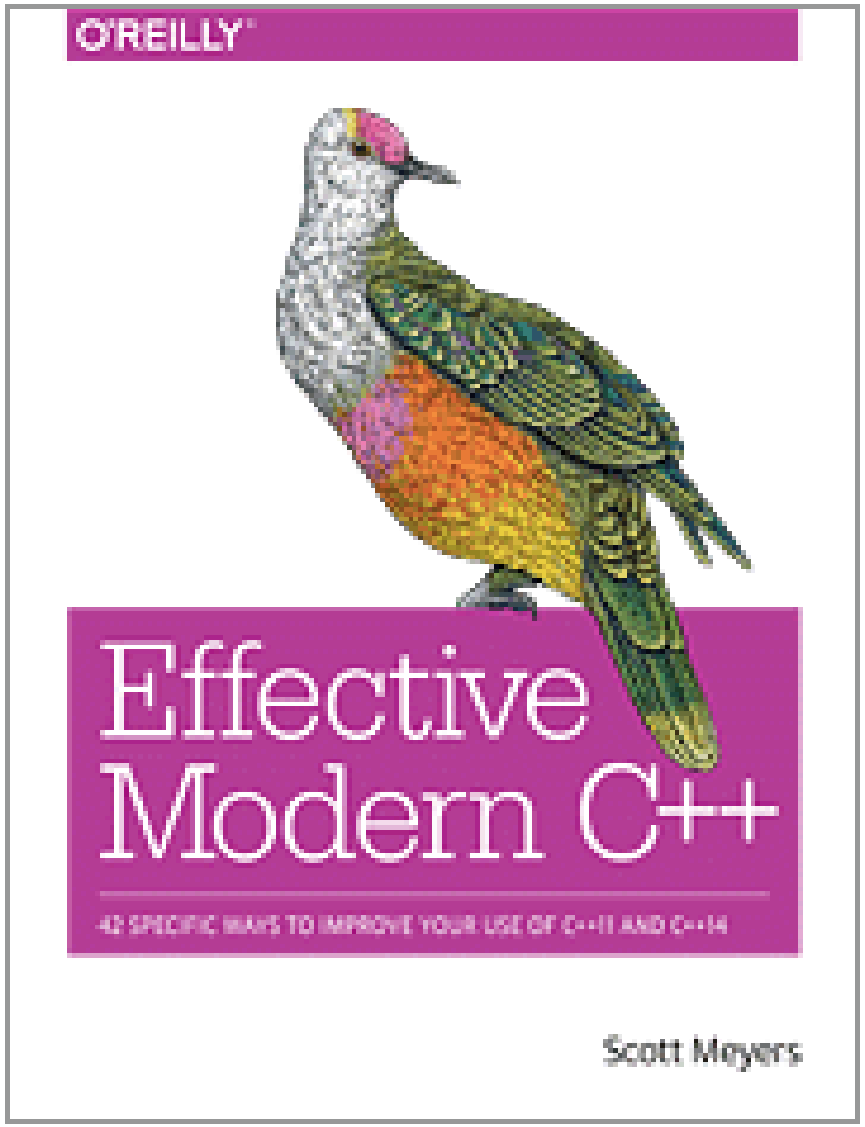
Evolution of C++ Language

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
• Templates • STL wit containers und algorithms • Strings • I/O Streams	• Move semantic • Unified initialisation • `auto` and `decltype` • Lambda functions • `constexpr` • Multithreading and the memory model • Regular expressions • Smart pointers • Hash tables • `std::array`	• Reader-writer locks • Generic lambda functions	• Fold expressions • constexpr if • Structured binding • `std::string_view` • Parallel algorithms of the STL • Filesystem library • `std::any`, `std::optional`, and `std::variant`	• Coroutines • ~~Contracts~~ • Modules • Concepts • Ranges library

Recommended Books and Sites

- **Effective Modern C++,**
  Scott Meyers

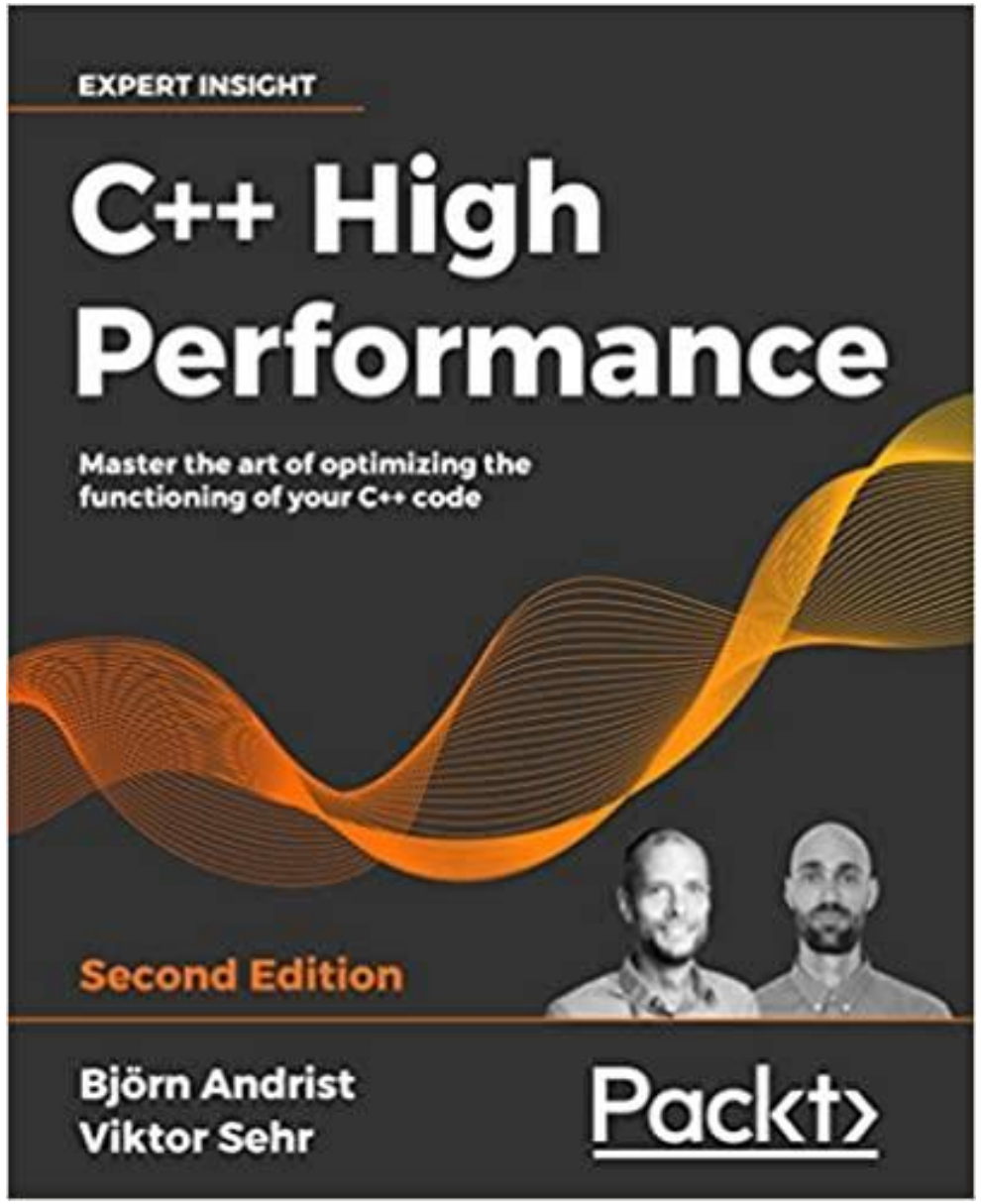
  O’Reilly



Recommended Books and Sites (Cont…)

- **C++ High Performance,**
  Björn Andrist and Viktor Sehr

  Packt



Recommended Books and Sites (Cont…)

- Udemy: "Complete Modern C++ (11/14/17)"
- O'Reilly site.
- Matt Godbolt : Why C++ isn't dead
- CppCon Channel on youtube
- C++ Weekly by Jason Turner on youtube

C++11 Language Features

Core language **runtime** performance enhancements

Core language **build-time** performance enhancements

Core language **functionality** improvements

Core language **usability** enhancements

auto & decltype keywords

auto

- **C++03 problem**: **(1)** we need too many time to write out things that the compiler already knows.
  Focus too much on types of variables and return types.
  Especially in STL containers, where it can be tedious…

```
std::map< std::string, std::vector< Widget > > table;

std::map< std::string, std::vector< Widget > >::iterator i = table.find("42");
```

auto (Cont…)

- **C++03 problem**: **(2)** Sometimes we cannot know the exact type of the return expression:

```
template<class T1, class T2>
T1 add1(const T1& a, const T2& b){
    return a + b; //always of type T1?
}
```

auto (Cont…)

- **C++11 solution**: use a sort of placeholder for a type - **auto**. The compiler will deduce the actual type of a variable that is being declared from its initializer.

```
int main()
{
    auto i = 42;          //i is an int
    auto l = 42LL;        //l is a long long
    auto p = new MyClass; //p is MyClass*
}
```

auto (Cont…)

- Using STL becomes much simpler:

```
std::map< std::string, std::vector< Widget > > table;

// C++98
std::map< std::string, std::vector< Widget > >::iterator i = table.find( "42" );

// Since C++11
auto j = table.find( "42" );
```

auto - decltype

- If the compiler can know at compile time the type of the returned expression, then we can use the `decltype` mechanism to introduce this type to the compiler.

```
template<class T1, class T2>
THE_TYPE_OF_A+B add1(const T1& a, const T2& b){
    return a+b; //always of type T1?
}
```

decltype (Cont…)

- **`decltype`** inspects the declared type of an entity or the type and value category of an expression.

- Syntax:

```
decltype ( entity )
decltype ( expression )
```

- `auto` lets you declare a variable with particular type whereas `decltype` lets you extract the type from the variable so `decltype` is sort of an operator that evaluates the type of passed expression.

decltype (Cont…)

```
int fun1() { return 10; }
char fun2() { return 'g'; }

int main(){
    // Data type of x is same as return type of fun1()
    // and type of y is same as return type of fun2()
    decltype(fun1()) x;
    decltype(fun2()) y;

    std::cout << typeid(x).name() << std::endl;
    std::cout << typeid(y).name() << std::endl;
}
```

```
int func(){
    return 0;
}
int func(int a){
    return 0;
}

int main(){
    int i = 4;

    decltype(func) var1;
           // Incorrect usage. func names an overload function

    // Correct usage. The overload operation is not ambiguous
    decltype(func(i)) var2;
}
```

decltype (Cont…)

- If the <object expression> or the <pointer expression> is specified with a constant or volatile qualifier, the type qualifier does not contribute to the result of `decltype(`*`expression`*`)`.

- The lvalueness or rvalueness of the object expression or the pointer expression does not affect whether `decltype(expression)` is a reference type or not.

decltype type deduction from r-value expressions

```
struct Foo{
    int _x;
};

int main(){
    struct Foo f;
    const struct Foo g = {0};
    struct Foo func();//function returning a Foo by value

    decltype(g._x) var1;          // int
    decltype(func()._x) var3;     // int
}
```

decltype and l-value expression

```
struct A {double x;};

int main(){
    A aObj;
    A* a = &aObj;

    decltype(a->x) y;   //y is double (declared type)
    y = 3.4;

    decltype((a->x)) z = y; //z is double& (lvalue expression)

    std::cout<<"y = "<<y<<" , z = "<<z<<std::endl;

    z += 1.5;

    std::cout<<"y = "<<y<<" , z = "<<z<<std::endl;
}
```

decltype and const expression

```
struct A {double x;};

int main(){
    A aObj;

    //------- declare a const pointer -------
    const A* ca = &aObj;

    decltype(ca->x) r;  //double

    decltype((ca->x)) s = r; //const double&

    //s += 1.5;   //error since s is const!
}
```

decltype (Cont…)

```
1. const int bar();
2. int i;
3. struct A { double x; };
4. const A* a = new A();

1. decltype(bar()) x2;
2. decltype(i) x3;
3. decltype(a->x) x4;
4. decltype((a->x)) x5 = x4;
```

What is the type of x2,x3,x4,x5?

decltype (Cont…)

```
1. const int bar();
2. int i;
3. struct A { double x; };
4. const A* a = new A();

1. decltype(bar()) x2;          // type is int
2. decltype(i) x3;              // type is int
3. decltype(a->x) x4;           // type is double
4. decltype((a->x)) x5 = x4; // type is const double&
```

decltype (Cont…)

```
template <typename T1, typename T2>
T2 add(const T1& a, const T2& b){
    return a+b;
}

int main(){
    std::cout<< add(1.8 , 2) << std::endl;
    std::cout<< add(2 , 1.8) << std::endl;
}
```

any problem?

decltype (Cont…)

```
template<class T1, class T2>
T1 add1(const T1& a, const T2& b){
    return a+b; //always of type T1?
}
```

C++11

```
template<class T1, class T2>
auto add2(const T1& a, const T2& b) -> decltype(a+b){
    return a+b;
}
```

decltype

```
/*
template <typename T1, typename T2>
T2 add(const T1& a, const T2& b){
    return a+b;
}
*/

template <typename T1, typename T2>
decltype(auto) add(const T1& a, const T2& b){
    return a+b;
}

int main(){
    std::cout<< add(1.8 , 2) << std::endl;
    std::cout<< add(2 , 1.8) << std::endl;
}
```

With C++14, the combination of `decltype` and `auto` is possible, and we can simply write `decltype(auto)` or just `auto`.

C++14

decltype (Cont…)

- We usually should use `auto` as the return type.
- We use `decltype(auto)` as the return type, when we forward a return value from another function and want to keep its type (usually with templates).

```
template < typename T >  //plain auto never deduces to a reference
auto foo( T& t ) {
    return t.value() ;
}

template < typename T >  //auto& always deduces to a reference
auto& bar( T& t ) {
    return t.value() ;
}

// return prvalue if t.value() is a r-value
// return lvalue if t.value is a l-value
// decltype(auto) has decltype semantics
//(without having to repeat the expression).
template < typename T >
decltype(auto) foobar( T& t ) {
    return t.value() ;
}
```

When should we use delctype(auto) as the return type instead of auto?

cont…

decltype (Cont…)

- What will the compiler say about the following code?

```
void testAutoDecltype(){
   struct A { int i = 0; int& value() { return i; } } a;
   struct B { int i = 0; int value() { return i; } } b;

   foo(a) = 20;
   foo(b);

   bar(a) = 20;

   bar(b);

   foobar(a) = 20;
   foobar(b);
}
```

```
void testAutoDecltype()
{
   struct A { int i = 0; int& value() { return i; } } a;
   struct B { int i = 0; int value() { return i; } } b;

   //foo(a) = 20; //Error: a.value() returns int& but foo(a) returns r-value.

   foo(b); //Fine: b.value() returns int and foo(b) is also int.

   //Fine: a.value() returns int& and bar's return value is auto& this int& (lvalue).
   bar(a) = 20;

   //Error: b.value() returns int and bar's return value is int&,
   //       but we cannot return reference to local variable
   //bar(b); //Error: auto& always deduces to a reference

   //the return value of foobar is always as the return type of t.value()
   foobar(a) = 20; // Fine: expression evaluates to lvalue of type int
   foobar(b);      // Fine: expression evaluates to prvalue of type int
}
```

decltype (Cont…)

- **C++98** had a single set of rules for type deduction: the one for function templates.
- **C++11** modifies that ruleset a bit and adds two more, one for `auto` and one for `decltype`.
- **C++14** then extends the usage contexts in which `auto` and `decltype` may be employed.

decltype (Cont…)

- When the template type deduction rules are applied in the context of auto, they sometimes seem less intuitive than when they're applied to templates.

- For that reason, **it's important to truly understand the aspects of template type deduction** that auto builds on.

decltype (Cont…)

- Similarly to the `sizeof` operator, the operand of `decltype` is unevaluated.

auto rules

- When using **auto**, the compiler **discards all qualifiers** and consider only the type.

```
int main()
{
    const int x = 2;
    auto var1 = x;    //var1 is "only" int, and not const int.
    var1++;
    const auto var2 = x;     //now var2 is a constant
    //var2++;    //ERROR!
}
```

auto rules (Cont…)

- using **auto&** **keeps the qualifiers**.

```
int main()
{
    const int x = 2;
    //-----------------------------------------
    // auto& - compiler keeps the qualifiers
    auto& var3 = x; //var3 is a const reference
    //var3++;   //ERROR!
}
```

auto (Cont…)

- auto becomes int type if even an integer reference is assigned to it.
- To make it reference type, we use `auto&`.

```
// function that returns a 'reference to int' type
int& fun() { static int x = 3; return x;  }

// m will default to int type instead of int& type
auto m = fun();

// n will be of int& type because of use of extra & with auto keyword
auto& n = fun();
```

auto (Cont…)

- Using **auto** with **pointers** also **keeps the qualifiers**.

```
const int x = 2;
auto ptr = &x;  //ptr is a const int*
//(*ptr)++; //ERROR!
```

auto and return value

- **C++11 problem**: `auto` cannot be used as a return type of a function.
  - However, it is possible to return an `auto` type while using the `decltype` keyword.

```
template <typename Builder>
auto makeAndProcessObject (const Builder& builder) -> decltype( builder.makeObject() )
{
    auto val = builder.makeObject();
    // do stuff with val...
    return val;
}
```

auto and return value (Cont…)

- In some cases, we cannot use `decltype` for the return value:

```
????? getLambda(){
    //lambda expression type is unpacified
    return ([](){std::cout<<"Hello World";});
}
```

- Lambda function is not an expression that can be written inside `decltype`.

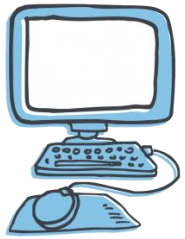
auto and return value (Cont…)

- **Since C++14** it is possible to write:

```
template <typename A, typename B>
auto do_something(const A& a, const B& b)
{
  return a.do_something(b);
}
```

- In C++14, the compiler deduces the return type of the methods that have "`auto`" as return type.

auto and return value (Cont…)



- Let’s convert the following code into C++14 style:

```
//C++98 style
struct Foo1{
    int val() const{
        return _m;
    }
    const int& cref() const{
        return _m;
    }
    int& mref(){
        return _m;
    }
private:
    int _m;
};
```

auto and return value (Cont…)

New Syntax using auto (C++14)	Old Syntax

`struct Foo{`
`  auto val() const{`
`  return _m;`
`  }`
`  auto& cref() const{`
`  return _m;`
`  }`
`  auto& mref(){`
`  return _m;`
`  }`
`private:`
`  int _m;`
`};`
	`struct Foo{`
	`  int val() const{`
	`  return _m;`
	`  }`
	`  const int& cref() const{`
	`  return _m;`
	`  }`
	`  int& mref(){`
	`  return _m;`
	`  }`
	`private:`
	`  int _m;`
	`};`

auto and return value (Cont…)

- Now we can return a lambda function using auto:

```
auto getLambda(){
    return ([](){std::cout<<"Hello World";});
}

int main()
{
    auto f = getLambda();
    std::cout<<"Lambda output: ";
    f();
    std::cout<<std::endl;
}
```

auto and return value (Cont…)

- All returned values must be of the same type.

```
auto f(int n)
{
    if (n == 1)
        return 1;
    return 2.0;
}   //error: cannot deduct the return type
```

auto& and const

```
struct Foo{
    const int& cref() const{
        return _m;
    }
    int& mref(){
        return _m;
    }
private:
    int _m = 0;
};

int main()
{
    Foo f;
    auto& cref = f.cref();
    //cref++;   //ERROR!
    auto& mref = f.mref();
    mref++;
}
```

auto& and const (Cont…)

- We can either use `auto&` or `const auto&` to declare a const reference variable:

```
int main(){
    auto foo = Foo{};
    auto& cref = foo.cref(); //cref is a const reference – bad practice!
    auto& mref = foo.mref(); //mref is a mutable reference
}
```

- Even though it is perfectly valid, it is preferred to always explicitly express that we are dealing with a const reference by using `const auto&`, and more important, we leave the `auto&` to only denote mutable references.

Declaring non-type template parameters with auto

- **C++17 addition**: we can use auto for deducing non-type template parameters.

```
template <typename T, T value> constexpr T c = val;
constexpr auto const IntConstant42 = c<int, 42>;
```

C++17

```
template <auto value> constexpr auto v = value;
constexpr auto const IntConstant42 = v<42>;
```

Declaring non-type template parameters with auto (Cont…)

- Example: declaring a compile-time list:

```
template <auto ... vs>
struct HeterogenousValueList {};

using MyList = HeterogenousValueList<'a',100,'b’>;
```

auto - Be careful!

- Using auto should be done very carefully:

```
std::deque<int> v = { .. };
for( int i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```

```
std::deque<int> v = { .. };
for( auto i = v.size()-1; i>0; i-=2) {
    f( v[i], v[i-1]);
}
```

any problem?

auto - Be careful! (Cont…)

- use carefully when:
  - the interface depends on an exact type
  - inference from a complex expression
  - computation/correctness is based on an exact type

```
for( auto i = 0; i < v.size(); ++i) ....
```

int i

wide type comparison

size_t
64-bit architectures:
unsigned long long

narrow type arithmetic

auto&& - Forwarding reference (Universal Reference)

- In contrast to the const reference (`const auto&`), `auto&&` allows you to mutate objects it references, include temporaries.

nullptr

nullptr

- **C++98 problem**: Zero used to be the value of null pointers, and that has drawbacks due to the **implicit conversion** to integral types.
- Oddly, in C++, the expression used, 0 (or NULL, always #defined to zero) was not even a pointer type.

```
void func(int n){}
void func(char* s){}

int main()
{
    func(NULL); //error: call of overloaded 'func(long long int)' is ambiguous
}
```

nullptr (Cont…)

- **C++11 solution**: `nullptr` is a new keyword that can (and should!) be used to represent NULL pointers.
  - The keyword `nullptr` denotes a value of **type** `std::nullptr_t` that represents the null pointer literal.
  - Implicit conversions exists from `nullptr` to null pointer value of any pointer type and any pointer-to-member types, but also to `bool` (as false).
  - But **no implicit conversion to integral types** exist.
  - For backward compatibility 0 is still a valid null pointer value.

nullptr (Cont…)

```
void foo(int* p){ std::cout<<"inside foo(int*)\n"; }
void bar(std::shared_ptr<int> p){
                std::cout<<"inside bar(shared_ptr<int>)";
}
int main(){
   func(nullptr);
   int x = NULL;
   //int y = nullptr; //error!

   int* p1 = NULL;
   int* p2 = nullptr;
   if(p1==p2){
      std::cout<<"p1 and p2 can be compared\n";
   }
   foo(nullptr);
   bar(nullptr);
}
```

nullptr (Cont…)

- `nullptr` is its own unique type, you can use it as a constructor or function argument when you want to be sure that you only ever take an empty pointer for a value.

```
void func(std::nullptr_t) {}
```

nullptr (Cont…)

```
void func(int n){ std::cout<<"inside func(int)\n"; }
void func(char* s){ std::cout<<"inside func(char*)\n"; }
void func(std::nullptr_t) {
                std::cout<<"inside func(nullptr)\n";
}
//C++11:
int main()
{
   func(nullptr); //prints: inside func(nullptr)
}
```

non-member begin & end

non-member begin() and end()

- **C++98 problem**: generic templated methods who need to iterate over a container, can handle only containers who support iterators.
- What if we send a c-style array?

non-member std::begin() and std::end()

```
template<class T>
void print_container_elements(T& data){
    for(auto itr = data.begin(); itr != data.end(); ++itr){
        std::cout<<*itr<<" ";
    }
    std::cout<<std::endl;
}

int main(){
    std::vector<int> vi { 1,2,3 };

    //printing all elements:
    print_container_elements(vi);

    int arr[] = { 1,2,3 };
    print_container_elements(arr);     //Error!
}
```

non-member std::begin() and std::end()

- **C++11 solution**: make `begin()` and `end()` global template methods, with specialization to basic types.

- Free-functions allow for more generic programming because they can be added afterwards, on a data-structure you cannot alter.

non-member std::begin() and std::end()

```
template<class T>
void print_container_elements(T& data){
    for(auto itr = std::begin(data); itr != std::end(data); ++itr){
        std::cout<<*itr<<" ";
    }
    std::cout<<std::endl;
}

int main(){
    std::vector<int> vi { 1,2,3 };

    //printing all elements:
    print_container_elements(vi);

    int arr[] = { 1,2,3 };
    print_container_elements(arr);
}
```

Range-based “for” loops

Range-based “for” loops

- **C++98 problem**: loop structure can be much simpler. Not always we are interested in the value of the iteration variable. Sometimes using the iterators is exhausting. **Other languages has foreach…**

```
int main(){
   std::map<int,std::string> names =
           {{1,"avi"}, {2,"gil"}, {3, "shira"}};
   for(auto itr = names.begin(); itr != names.end(); ++itr){
      std::cout<<itr->second<<" ";
   }
   std::cout<<std::endl;
}
```

range-based “for” loops (Cont…)

- **C++11 solution**: range-based loop, like the "foreach" paradigm of iterating over collections.
  - In the new form, it is possible to iterate over C-like arrays, initializer lists and anything for which the non-member `begin()` and `end()` functions are overloaded.
  - This for each for is useful when you just want to get and do something with the elements of a collection/array and don't care about indexes, iterators or number of elements.

Range-based "for" loops (Cont…)

```
int main(){
   //old style for loop
   std::map<int,std::string> names =
           {{1,"avi"}, {2,"gil"}, {3, "shira"}};
   for(auto itr = names.begin(); itr != names.end(); ++itr){
      std::cout<<itr->second<<" ";
   }
   std::cout<<std::endl;

   //C++11 style:
   for(auto i : names){
      std::cout<<i.second<<" "; //i is std::pair<int,std::string>
   }
   std::cout<<std::endl;
}
```

Range-based "for" loops (Cont…)

- range based loop with `std::initializer_list`:

```
int main()
{
   for(auto i : {1,2,3,4}){
      std::cout<< i*i << " ";
   }
   std::cout<<std::endl;
}
```

Range-based for loop: Exercise

- **Exercise:** open file range_based_for_exercise.cpp from git and implement the output requirements.

```
int main()
{
    std::map<std::string,std::vector<std::string>> foodNames =
    {
        {"drinks",{"water", "milk", "juice"}},
        {"fruits",{"apple", "grapes"}},
        {"vegetables",{"carrot", "potato", "cucumber"}}
    };

    //use range-based for loop to print all data to screen:
    // drinks: water milk juice
    // fruits: apple grapes
    // vegetables: carrot potato cucumber
}
```

Uniform Initialization

Uniform Initialization

- **C++98 problem**: Initialization in C++98 has **four** different initialization notations.
  - The user need to remember the distinctions between initializing fundamental types, aggregate and non-aggregate types, and arrays and standard containers.

```
int main(){
   int i;    // uninitialized built-in type
   int j=10;  // initialized built-in type
   int k(10); // initialized built-in type
   int a[]={1, 2, 3, 4}; // Aggregate initialization
    X x1;   // default constructor
   X x2(1);  // Parametrized constructor
    X x3=3; // Parametrized constructor with single argument
   X x4=x3; // copy-constructor
}
```

Uniform Initialization (Cont…)

- In C++ we have two ways to initialize an object:
  - **direct initialization** - Initializes an object from explicit set of constructor arguments.
  - **copy initialization** - Initializes an object from another object.

```
std::string s1("test");   // direct initialization
std::string s2 = "test";  // copy initialization
```

Uniform Initialization (Cont…)

- A little rehearsal about function resolution:

```
void f(int&){std::cout<<"f(int&)\n";}

void f(const int&){std::cout<<"f(const int&)\n";}

int main() {
   int i = 1;

   f(i);    // calls f(int&)
}
```

Uniform Initialization (Cont…)

```
struct X;
struct Y { operator X(); };

struct X {
  X() { }
  X(const Y&) { std::cout << "Direct initialization" << '\n'; }
  X(const X&) { std::cout << "Copy ctor" << '\n'; }
};

Y::operator X() { std::cout << "Copy initialization" << '\n’;
                  return X();
                }

int main() {
   Y y;
  X x1 = y;
}
```

Uniform Initialization (Cont…)

- **C++11 solution**: use a uniform way to initialize all types, using the { } braces.

```
int main(){
    int a{0};

    std::string s{"hello"};
    std::string s2{s}; //copy constructor

    std::vector<std::string> vs{"alpha", "beta", "gamma"};

    std::map<int,int> sources {{1,58}, {2,79}};

    double* pd = new double[3] {0.5, 1.2, 12.99};
}
```

Uniform Initialization (Cont…)

- To uniformly initialize objects regardless of their type, use the brace-initialization form {} that **can be used for both direct initialization and copy initialization**.
  - When used with brace initialization, these are called direct list and copy list initialization.

```
T object {};     //value initialization
T object {other};   // direct list initialization
T object = {other}; // copy list initialization
```

Uniform Initialization (Cont…)

- **C++98 problem**: initialization is complicated in some cases:
  - No way to initialize a member **array** (except using the default ctor).
  - No convenient form of initializing **containers**.
  - No way to initialize **dynamically allocated POD** types (POD stands for Plain Old Data - that is, a class (whether defined with the keyword struct or the keyword class) without constructors, destructors and virtual members functions).

Uniform Initialization (Cont…)

```
//No way to initialize a member array
//( except using the default ctor).
class A {
public:
    A(int x) : _x(x) {}
private:
    int _x;
};

int main(){
 //   A array[10];
}
```

Uniform Initialization (Cont…)

```
//No convenient form of initializing containers.
int main()
{
    //we can initialize static vector:
    int arr[] = {0,1,2,3,4,5,6,7,8,9};

    //but we cannot do it with containers:
    std::vector<int> vec;
    for(int i=0; i<10; i++){
        vec.push_back(i);
    }
}
```

Uniform Initialization (Cont…)

```
//No way to initialize dynamically allocated POD types.
int main()
{
    char *buff=new char[1024];
            //no proper way to initialize the elements of buff
}
```

Uniform Initialization (Cont…)

- **C++11 solution**: with brace initialization we can initialize all those cases at the same way:

```
int a{0};

std::string s{"hello"};
std::string s2{s}; //copy construction

std::vector <std::string> vs{"alpha", "beta", "gamma"};
std::map<int, int> scores { {1,58}, {2,79}};

double *pd= new double [3] {0.5, 1.2, 12.99};

class C{
    int x[4];
public:
    C(): x{0,1,2,3} {}
};
```

Uniform Initialization (Cont…)

- With braced initialization, it is easier to see when you are constructing an object or calling a function.
- It increases concision.

```
struct blah{
   // will use a default constructed string if none is provided
   explicit blah(const std::string & s = {}) : _s{s} {}
   // much shorter than
   // explicit blah(const std::string & s = std::string()) : _s(s) {}
private:
   std::string _s;
};
```

Uniform Initialization (Cont…)

- Brace initialization does not allow narrowing.
  - An integer cannot be converted to another integer that cannot hold its value.
    - For example, char to int is allowed, but not int to char.
  - A floating-point value cannot be converted to another floating-point type that cannot hold its value.
    - For example, float to double is allowed, but not double to float.
  - A floating-point value cannot be converted to an integer type.
  - An integer value cannot be converted to a floating-point type.

Uniform Initialization (Cont…)

```
void fun(double dVal, int iVal2)
{
    int x2 = dVal;
    char c2 = iVal2;
    int x3 {dVal};
    char c3 {iVal2};
    char c4 {24};
    char c5 {264};
    int x4 {2.0};
}
```

What will the compiler say about this code?

Uniform Initialization (Cont…)

```
void fun(double dVal, int iVal2)
{
    int x2 = dVal;       // if val==7.9, x2 becomes 7 (bad)
    char c2 = iVal2;     // if val2==1025, c2 becomes 1 (bad)
    int x3 {dVal};       // error: possible truncation (good)
    char c3 {iVal2};     // error: possible narrowing (good)
    char c4 {24}; // OK: 24 can be represented exactly as a char (good)
    char c5 {264};       // error (assuming 8-bit chars): 264 cannot be
                         // represented as a char (good)
    int x4 {2.0}; // error: no double to int value conversion (good)
}
```

Uniform Initialization (Cont…)

- Notice that the following code will generate a compilation error, just like with old array aggregate initialization:

```
int main()
{
   int arr[5]{1,2,3,4,5,6};
           // error: too many initializers for 'int [5]'
}
```

Default Initialization using { }

- An empty pair of braces indicates **default initialization**.
- Default initialization of POD types usually means initialization to binary zeros, whereas for non-POD types default initialization means default construction.

```
int main(){
    int n{};          //zero initialization: n is initialized to 0
    int *p1{};        //initialized to nullptr
    double d{};       //initialized to 0.0
    char s1[12]{};    //all 12 chars are initialized to '\0’
    std::string s2{};   //same as: string s;
    char *p2 = new char [5]{}; // all five chars are initialized to '\0'
}
```

Default values of member variables

- **since C++11** it is possible to give default values inside the class declaration:

```
class C
{
    int _x=7; //class member initializer
public:
    C();
};
```

Default values of member variables (Cont…)

- Works also for member arrays:

```
class C2
{
    int _y[5] {1,2,3,4};
public:
    C2();
};
```

Default values of member variables (Cont…)

- A class member initializer can consist of the traditional equal sign or the new brace-init:

```
class C3{
    std::string _s1{"abc"};
    std::string _s2 = std::string("def");
    double      _d {0};
    char *      _p {nullptr};
    int         _y[5] {1,2,3,4};
    int *       _q = new int[10];
public:
    C3();
};
```

Default values of member variables (Cont…)

- Notice that the capacity of using braces to specify default initialization values for non-static data members is shared with the “=“ initialization syntax, but not with parentheses:

```
class Widget {
  …

private:
  int x{ 0 }; // fine, x's default value is 0
  int y = 0;  // also fine
  int z(0);   // error!
};
```

Default values of member variables (Cont…)

- On the other hand, uncopyable objects (e.g., std::atomics) may be initialized using braces or parentheses, but not using “=”:

```
class A{
public:
    A() { std::cout<<"default ctor\n";}
    A(int) {std::cout<<"A(int) ctor\n";}
    A(const A&) = delete; //{std::cout<<"copy ctor\n";}
};

int main(){
    A a1{0};
    A a2 = 0;//error! use of deleted function 'A::A(const A&)'
}
```

Default values of member variables (Cont…)

- Regardless of the initialization form used, the compiler conceptually transforms every class member initializer into a corresponding mem-init. Thus, class C3 above is semantically equivalent to the following class:

```
class C3{
    std::string _s;
    double      _d;
    char *      _p;
    int         _y[5];
public:
    C3() : _s("abc"), _d(0.0), _p(nullptr), _y{1,2,3,4} {}
};
```

Default values of member variables (Cont…)

- If the same data member has both a class member initializer and a mem-init in the constructor, the latter takes precedence.

```
class C5{
    int _x=7;    //class member initializer
public:
    C5() {} //x is initialized to 7 when the default ctor is invoked
    C5(int y) : _x(y) {} //overrides the class member initializer
};

C5 c;        //c._x = 7
C5 c2(5);    //c2._x = 5
```

Trivial type

- A **trivial type** is a type that follows the following rules:
  - has compiler-generated constructors
  - no virtual member functions
  - no virtual base classes
  - no non-static members with default initializers
  - base classes and member objects are also trivial.

MKBN

Standard layout type

- A class is **standard-layout** when all non-static data members have the same access control, and the following conditions are satisfied:
  - All non-static data members have the same access control.
  - Has no virtual functions or virtual base classes.
  - Has no non-static data members of reference type.
  - All non-static data members and base classes are themselves standard layout types.
  - Has no two (possibly indirect) base class subobjects of the same type (diamond).
  - Has all non-static data members and bit-fields declared in the same class (either all in the derived or all in some base)

aggregate class

- A **class** is an **aggregate** if:
  - All of its data members are public
  - It does not define any constructors
  - It has no in-class initializers
  - It has no base classes or virtual functions

- **An aggregate object can be initialized with brace initialization syntax**.

POD type

- **POD** type is a type that consists of nothing but plain old data.
- This means the type is compatible with the types used in the C programming language, that is, can be exchanged with C libraries directly, in its binary form.
  - scalar type
  - trivial type
  - is a standard layout type
  - all non-static members are POD

MKBN

POD type (Cont…)

- It is safe to copy POD objects with memcpy:

```
#define N sizeof(T)
char buf[N];
T obj;                  // obj initialized to its original value
memcpy(buf, &obj, N); // between these two calls to memcpy,
                        // obj might be modified
memcpy(&obj, buf, N); // at this point, each subobject of obj of
                        // scalar type holds its original value
```

POD type (Cont…)

```
struct X {
    X() = default;
};

struct Y {
    Y() { };
};

void test1() {
    static_assert(std::is_trivial<X>::value, "X should be trivial");
    static_assert(std::is_pod<X>::value, "X should be POD");

    static_assert(!std::is_trivial<Y>::value, "Y should not be trivial");
    static_assert(!std::is_pod<Y>::value, "Y should not be POD");
}
```

aggregate initialization

- Aggregate initialization can be performed only on aggregate classes:

```
struct S{ int i; };

class A_NotAggregate{
public:
    virtual void f() {} //no virtual functions in aggregate classes
//private:   //no private data members
    int x;
};

int main(){
    S s{1};

    A_NotAggregate a{1};  //ERROR!
}
```

Initializer Lists

- **C++11** supports initializing aggregated types by using the new templated type `std::initializer_list`.
  - you need to include the header <initializer_list>.
- It is a proxy object that represents an array of objects.
- Constructed automatically from a braced-list of elements.
- All STL containers were getting a new constructor for supporting such initialization.

```cpp
template <class T, int Size>
class Pool{
public:
    Pool() = default;
    Pool(std::initializer_list<T> values){
        if(values.size()<Size){
            for(auto it : values){
                add(it);
            }
        }
    }
    void add(T val){
        if(_size<Size){
            _data[_size++] = val;
        }
    }
    void remove(){
        if(_size>0){
            --_size;
        }
    }
    T operator[] (int index){
        assert(index>=0 && index<Size);
        return _data[index];
    }

    int getSize() const {
        return _size;
    }
private:
    T _data[Size];
            //static array of T elements
    int _size{};    //initialized to zero
};
```

```
int main(){
    //----without initialization list we can initialize Pool only using add
    Pool<int,10> thePool;
    thePool.add(1);
    thePool.add(2);
    thePool.add(3);
    //printing thePool:
    for(int i=0; i<thePool.getSize(); ++i){
        std::cout<<thePool[i]<<" ";
    }
    std::cout<<std::endl;

    //----with initialization list constructor
    Pool<int,10> anotherPool{1,2,3};
    //printing thePool:
    for(int i=0; i<thePool.getSize(); ++i){
        std::cout<<thePool[i]<<" ";
    }
    std::cout<<std::endl;

    //we could also use:
    auto values = {1,2,3,4};
    Pool<int,10> anotherPool2(values);
}
```

Initializer Lists (Cont…)

- Suppose your code contains the following declaration of a vector:

```
std::vector<double> vd {0.5, 1.33, 2.66};
```

- **The compiler detects the initializer list {0.5, 1.33, 2.66} and performs the following steps**:
  1. **Detect the type of the values** in the initializer list. In the case of {0.5, 1.33, 2.66}, the type is double.
  2. **Copy** the values from the list into an array of three doubles.
  3. **Construct** an `initializer_list<double>` object that "wraps" the array created in the preceding step.
  4. **Pass** the `initializer_list<double>` object by reference to vd's sequence constructor. The constructor in turn allocates n elements in the vector object, initializing them with the values of the array.

Initializer Lists (Cont…)

- The only situation where = is preferred over { } is when using auto keyword to get the type determined by the initializer.

```
auto z1 {99}; // z1 is an initializer_list<int>
auto z2 = 99; // z2 is an int
```

Initializer Lists (Cont…)

- **C++17 has new rules for list initialization**, differentiating between the direct- and copy-list-initialization.

- **for copy list initialization:** `auto` deduction will deduce a `std::initializer_list<T>` if all elements in the list have the same type, or be ill-formed.
- **for direct list initialization:** `auto` deduction will deduce a `T` if the list has a single element, or be ill-formed if there is more than one element.

Initializer Lists (Cont…)

```
//until C++17:
auto a = {42};   // std::initializer_list<int>
auto b {42};     // std::initializer_list<int>
auto c = {4, 2}; // std::initializer_list<int>
auto d {4, 2};   // std::initializer_list<int>

//since C++17:
auto a = {42};   // copy init: std::initializer_list<int>
auto b {42};     // direct init: int
auto c = {4, 2}; // copy init: std::initializer_list<int>
auto d {4, 2};   // direct init: compilation error
```

Object construction improvement – Delegated Constructor

Object construction improvement

- **C++98 problem**: multiple code for the same initialization.

```
class MyClass
{
public:
    MyClass() : _val1(0), _val2(0){}    //ctor #1
    MyClass(int v1, int v2) : _val1(v1), _val2(v2) {}   //ctor #2

private:
    int _val1;
    int _val2;
};
```

both ctor's needs to initialize each member data.

Object construction improvement (Cont…)

- **C++98 solution:** declare a helper function and call it from both constructors.

```
class MyClass{
public:
    MyClass() { initDataMembers(0,0); }    //ctor #1
    MyClass(int v1, int v2) { initDataMembers(v1,v2); }   //ctor #2
protected:
    void initDataMembers(int v1, int v2){
        _val1 = v1;
        _val2 = v2;
    }
private:
    int _val1;
    int _val2;
};
```

what about object members? consts? references?

Object construction improvement (Cont…)

- **C++11 solution**: **constructor delegation**. One constructor can call another constructor of the same class.

```
class MyClass3
{
public:
   MyClass3() : MyClass3(0,0) {}    //ctor #1 calls constructor #2
   MyClass3(int v1, int v2) : _val1(v1), _val2(v2) {}   //ctor #2
private:
   int _val1;
   int _val2;
};
```

defaulted and deleted member functions

defaulted and deleted member functions

- **Problem**: if we can in C++11 to initialize non static data members within the class declaration, then we do not need the default constructor anymore, but if will not declare it, a compilation error may be generated:

```
class Point{
public:
    Point(int x, int y) : _x(x), _y(y) {}
private:
    int _x{0};
    int _y{0};
};

int main(){
    Point p;    //compilation error
}
```

defaulted and deleted member functions (Cont…)

- **C++11 solution**: declare the copy constructor as default with `=default` syntax. The compiler will generate the implicit default constructor even if other constructors are present.

```
class Point {
public:
   Point() = default;
   Point(int x, int y) : _x(x), _y(y) {}
private:
   int _x{0};
   int _y{0};
};
```

defaulted and deleted member functions (Cont…)

- The `=default` syntax can be used **only with methods that the compiler can synthesize** if we do not declare them:
  - default constructor.
  - copy constructor.
  - assignment operator.

- All other methods, including non default constructors or constructor with single argument with a default value, **cannot be defaulted.**

defaulted and deleted member functions (Cont…)

- **Why not just declaring `Foo(){}` instead of `Foo()=default` ?**
- Non-default implementations are non-trivial, which affects type semantics, e.g. makes a type non-POD.

defaulted and deleted member functions (Cont…)

```
struct Q{
    Q(const Q&) = default;  //Q is still aggregate class

    int x;
};

struct F{
    F(const F& f){} //F is not aggregate class anymore!

    int x;
};

int main(){
    Q q{5};
    // F f1{3};  //F is not aggregated thus this initialization
                 // is not allowed
}
```

defaulted and deleted member functions (Cont…)

- **C++98 problem**: suppose we want to declare a non-copyable object. This is the only option we had:

```
struct A
{
  A() {};

private:
  A(const A&);
  A& operator=(const A&);
};
```

defaulted and deleted member functions (Cont…)

- **There are several problems with this solution:**
  - We must declare the copy constructor as private ⇒ **no default constructor is generated by the compiler**.
  - Even if the explicitly-defined default constructor does nothing, it's considered non-trivial by the compiler. **It's less efficient than an automatically generated default constructor and prevents the objects from being a true POD type**.
  - The member functions and friends of this class **can still see and call them**.
  - **The intent is not clear** unless you understand all of the rules for automatic generation of the special member functions.

defaulted and deleted member functions (Cont…)

- **C++11 solution**: use the `=delete` syntax to force the compiler not to generate this method at all.
- This syntax can be applied on all class methods as well as non-member functions.

```
struct A{
   A() {}
private:
   A(const A&) = delete;
   A& operator=(const A&) = delete;
};
```

defaulted and deleted member functions (Cont…)

- Deleting of special member functions provides a cleaner way of preventing the compiler from generating special member functions that we don't want.
- Deleting of normal member function or non-member functions **prevents problematic type promotions** from causing an unintended function to be called.

```
void process(int data){}
void process(double data) = delete;
                    //prevents from double's to be sent to print(int)

int main(){
    // process(3.4); //ERROR!
}
```

Tuples and Structural Binding

Tuples and Structural Binding

- **C++ problem**: It is possible to return only one value from a function.

- **Simple Solution (#1):** We can use "sending by reference" mechanism in order to "return" more than one value:

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

void readFromDB( DbHhandler* h, std::string& rs,
                 bool& valid) {}
void process(std::string& str){}

int main(){
   DbHhandler db;
   std::string result;
   bool bValid;
    readFromDB(&db, result, bValid);
   if(bValid){
     process(result);
   }
}
```

non-intuitive in/out par

Returning multiple values from a function (Cont…)

- Maybe we can return one by value and send the others by reference?!

```
bool readFromDB(DbHhandler* h, std::string& rs){return true;}

int main()
{
    DbHhandler db;
    std::string result;

    if(readFromDB(&db,result)){
        process(result);
    }
}
```

non-intuitive separation

Returning multiple values from a function (Cont…)

- Keeping the Boolean inside and query it separately?

```
const std::string& readFromDB(DbHhandler* h);
bool result_ok(const DbHhandler* h);

int main()
{
    DbHhandler db;
    const std::string& rs = readFromDB(&db);
    if(result_ok(&db)){
      process(rs);
    }
}
```

additional fnc call
static internal state !!!

Returning multiple values from a function (Cont…)

- So, Let's use STL for that:

```
std::pair<std::string, bool>  readFromDB(DbHhandler* h);

int main()
{
    DbHhandler db;
    std::pair<std::string, bool> rs =  readFromDB(&db);
    if(rs.second){
      process( rs.first);
    }
}
```

unnecessary type verbose

Returning multiple values from a function (Cont…)

- **C++11** is here to help! we'll use **auto** to make it simple!

```
std::pair<std::string, bool> read( DbHhandler* h);

int main()
{
    DbHhandler db;
    auto rs = read(&db);
    if( rs.second){
      process(rs.first);
    }
}
```

What if we want to return more than two results?

What is first? second?
What if we already have s and valid variables?

Returning multiple values from a function (Cont…)

- **C++11 solution**: use **`tuple`** types.

- A tuple is an object that can hold several elements. The elements can be of different data types.
  The elements of tuples are initialized as arguments in order in which they will be accessed.

Returning multiple values from a function (Cont…)

- **Why not simply returning a struct with all data**?

- We can.

- There are performance issues. According to this link, std::tuple is much faster than a default struct.

- Since `std::tuple` is a standard type, it may be more optimized. On the other way, the struct's layout may be better and compatabily.

std::tuple

- **Tuple** – an object that can hold several elements.

- Has the following operations:
  - **get**() – access the tuple values and modify them, according to the index its gets.
  - **make_tuple**() – used to assign a tuple with values.
  - **tuple_size**() – returns the number of elements in the tuple.
  - **swap**() – swap the elements of two different same-type tuples.
  - **tie**() – unpack the tuple values into separate variables.
  - **tuple_cat**() – concatenates two tuples and returns a new tuple.

Tuple (Cont…)

```
int main()
{
    std::tuple<char,int,float> data(20,'g',17.5);

    std::cout<<"the size of the tuple is: "<<
            std::tuple_size<decltype(data)>::value <<std::endl;

    return 0;
}
```

Tuple (Cont…)

```
int main(){
    int i;
    char c;
    float f;

    std::tuple<int,char,float> tup1(20,'g',17.5);

    std::tie(i,std::ignore,f) = tup1;

    std::cout<< " Values of tuple are: "<<i<<","<<c<<","<<f<<
                std::endl;

}
```

Tuple (Cont…)

```
int main() {
   std::tuple <char, int, float> data; //all values are zero

   data = std::make_tuple('a', 10, 15.5); //init the tuple

   std::cout << "The initial values of tuple are : ";
   std::cout << std::get<0>(data) << " " << std::get<1>(data);
   std::cout << " " << std::get<2>(data) << std::endl;

   std::get<0>(data) = 'b';
   std::get<2>(data) = 20.5;

   std::cout << "The modified values of tuple are : ";
   std::cout << std::get<0>(data) << " " << std::get<1>(data);
   std::cout << " " << std::get<2>(data) << std::endl;

   }
```

Returning multiple values from a function (Cont…)

Tuple (Cont…)

```
int main(){
    std::tuple <int,char,float> tup1(20,'g',17.5);
    std::tuple <int,char,float> tup2(30,'f',10.5);

    auto tup3 = std::tuple_cat(tup1,tup2); //size of tup3 is 6!

    std::cout<<"The size of the new tuple is: ";
    std::cout<< std::tuple_size<decltype(tup3)>::value <<std::endl;
}
```

Tuple (Cont…)

- Let's go back to our example:

```
auto readFromDB_5( DbHhandler* h){
  std::string s;
  bool b;
  /*read data from db into s, using the handler, saving the status in b...*/
  return make_tuple(s,b);
}

int main(){
  std::string s; bool valid;
  tie(s,valid) = readFromDB(&db);
  if( valid){
    process( s);
  }
}
```

C++14

must be visible
for type deduction

Tuple (Cont…)

- Another way:

```
std::tuple<std::string,bool> readFromDB( DbHandler* h);

int main()
{
   auto rs = readFromDB(& db);
   f(rs);  //forward the result
   if( std::get<1>(rs)){
     doit( std::get<0>(rs));
   }
}
```

Structure Binding – C++17

- Structured Bindings give us the ability to declare multiple variables initialized from a tuple or struct.
- The main purpose of Structured Bindings in C++ 17 is to make the code clean and easy to understand.

Structure Binding – C++17 (Cont…)

```
struct Point {
      int x;
      int y;
};

int main() {
      Point p = {1, 2};

      int x_coord = p.x;
      int y_coord = p.y;

      std::cout << "X Coordinate : " << x_coord << std::endl;
      std::cout << "Y Coordinate : " << y_coord << std::endl;

      return 0;
}
```

C++98

C++11/14

```
struct Point
{
        int x, y;
        Point() : x(0), y(0) { }
        Point(int x, int y) : x(x), y(y)  { }
        auto operator()() {
                // returns a tuple to make it work with std::tie
                return std::make_tuple(x, y);
        }
};

int main() {
        Point p = {1, 2};
        int x_coord, y_coord;
        std::tie(x_coord, y_coord) = p();

        std::cout << "X Coordinate : " << x_coord << std::endl;
        std::cout << "Y Coordinate : " << y_coord << std::endl;
}
```

C++17

```
struct Point
{
        int x;
        int y;
};

int main( )
{
        Point p = { 1,2 };

        // Structure binding
        auto[ x_coord, y_coord ] = p;

        std::cout << "X Coordinate : " << x_coord << std::endl;
        std::cout << "Y Coordinate : " << y_coord << std::endl;

        return 0;
}
```

Structure Binding – C++17 (Cont…)

- Structured Binding can be used with arrays to get the elements from the array.
- The number of identifiers in the identifier list must be equal to the number of elements in the array.

```
int main(){
      int arr[3] = { 1, 2, 3 };

      auto[x, y, z] = arr;

      std::cout << x << " " << y << " " << z << std::endl;
      return 0;
}
```

Returning multiple values from a function (Cont…)

Structure Binding – C++17 (Cont…)

- This concept is not new in itself. Previously, it was always possible to return multiple values from a function and access them using std::tie.

```
std::tuple<char, int, bool> mytuple(){
    char a = 'a'; int i = 123; bool b = true;
    return std::make_tuple(a, i, b);
}

int main(){
    char a;
    int i;
    bool b;

    std::tie(a, i, b) = mytuple();
}
```

C++17

```
int main(){
    auto [a, i, b] = mytuple();
}
```

Structure Binding – C++17 (Cont…)

- Back to our example:

```
std::tuple<std::string,bool> readFromDB( DbHandler* h);

int main()
{
    auto [rs,valid] = readFromDB(&db);
    if(valid){
      process(rs);
    }
}
```

For more details about structural biding: P0144R0

Hands-On Exercise

- Add a loop to the following code that prints all pairs in the map:

```
int main(){
   // Creating a map with key and value
   // fields as String
   map<std:: string, std:: string> sites;

   sites.insert({ "GeeksforGeeks", "Coding Resources" });
   sites.insert({ "StackOverflow", "Q-A type" });
   sites.insert({ "Wikipedia", "Resources + References" });

   //TODO: use structured binding to print all keys & values in map
}
```

template & namespace aliases

template aliases

- **C++98 problem**: we cannot template the `typedef` and need to declare a new typedef for each `T`.

```
using ListsPool_Strings =  std::vector<std::list<std::string>>;
using ListsPool_ints =     std::vector<std::list<int>>;
```

template aliases (Cont…)

- **C++98 partial solution**: use a templated struct that wraps the `typedef`:

```
template <class T>
struct ListsPool{
    typedef std::vector<std::list<T>> type_t;
};

void test(){
    ListsPool<int>::type_t poolOfInts;
}
```

template aliases (Cont…)

- Not always intuitive…

```
template<typename T>
class Widget
{                                          // Widget<T> contains
private:                                   // a MyAllocList<T>
  typename MyAllocList<T>::type list;      // as a data member
  //...
};
```

Further reading: Effective Modern C++ item 9

template aliases (Cont…)

C++11 solution: **alias template**. Unlike the typedef, the alias-declarations can be templatized. With alias templates, we can define generic aliases and specialize them further:

```
template<class T>
using ListsPool_t = std::vector<std::list<T>>;

class MyClass{};

int main(){
    ListsPool_t<MyClass> poolOfMyClass;
}
```

```
template<class T>
using Pool_t = std::map<T, std::vector<T>>; // type-id is std::map<int,std::vector<int>>

Pool_t<int> v; // Pool_t<int> is the same as std::map<int,std::vector<int>>
```

template aliases (Cont…)

```
template <class T>
using ObjCounter_t = std::map<T,size_t>;

int main()
{
    ObjCounter_t<std::string> wordsCounter;
    wordsCounter["hello"]++;

    return 0;
}
```

template aliases (Cont…)

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

template <typename T>
using vec_t = std::vector<T, custom_allocator<T>>;

vec_t<int>          vi;
vec_t<std::string>  vs;
```

```
const char* getErrorMessage(int errorNum){
    //... implement creating the right message according to errorNum
    return "Testing...\n";
}

//v1
void printErrorMsg_v1(const char *(*pfn)(int), int errorNum){
    std::cout<<"ERROR: "<<(*pfn)(errorNum)<<std::endl;
}

//v2
typedef const char* (*PFN_t)(int);
void printErrorMsg_v2(PFN_t pfn, int errorNum){
    std::cout<<"ERROR: "<<(*pfn)(errorNum)<<std::endl;
}

//v3
using PFN = const char* (*)(int);
void printErrorMsg_v3(PFN pfn, int errorNum){
    std::cout<<"ERROR: "<<(*pfn)(errorNum)<<std::endl;
}
```

namespace alias

- Namespace aliases allow the programmer to define an alternate name for a namespace.
  - They are commonly used as a convenient shortcut for long or deeply-nested namespaces.

```
namespace foo {
    namespace bar {
        namespace baz {
            int qux = 42;
        }
    }
}

namespace fbz = foo::bar::baz;

int main(){
    std::cout << fbz::qux << '\n';
}
```

Override and final

Override and final

- **C++98 problem**: The `virtual` keyword is optional and that makes reading code a bit harder, because you may have to look through the top of the hierarchy to check if the method is virtual.

- In addition, declaring the method as virtual doesn't guarantee that it is an override one.

Override and final (Cont…)

```
class B{
public:
    virtual void f(short) { std::cout<< "B::f" << std::endl; }
};

class D : public B{
public:
    virtual void f(int) { std::cout<<" D::f" <<std::endl; }
};

int main(){
    B* pB = new D;
    pB->f(5);       //what will be printed?
}
```

Override and final (Cont…)

- **C++11 solution**: Two new special identifiers (not keywords) have been added:
  - `override`, to indicate that a method is supposed to be an override of a virtual method in a base class,
  - `final`, to indicate that a derived class shall not override a virtual method.

```
class B{
public:
    virtual void f(short) { std::cout<< "B::f" << std::endl; }
};

class D : public B{
public:
    virtual void f(int) override { std::cout<<" D::f" <<std::endl; }
        //compilation error! 'virtual void D::f(int)' marked 'override', but does not ov
erride
};
```

Override and final (Cont…)

- When used in a **virtual function** declaration or definition, `final` ensures that the function is virtual and specifies that it may not be overridden by derived classes.
- When used in a **class** definition, `final` specifies that this class may not appear in the base-specifier-list of another class definition (in other words, cannot be derived from).
- `final` is an identifier with a special meaning when used in a member function declaration or class head. In other contexts it is not reserved and may be used to name objects and functions.

Override and final (Cont…)

```
struct Base{
    virtual void foo();
};

struct A : public Base{
    void foo() final; //A::foo is overridden, and it is the final override
    void bar() final; //error: non virtual function cannot be overridden or be final
};

struct B final : public A{
    void foo() override;    //Error: foo cannot be overridden as it's final in A
};

struct C : public B{    //error: B is final
};
```

```
class FooInterface{
public:
    virtual void mainFlow() = 0;
protected:
    virtual void mainFlowImplementation() = 0;
};

class FooBase : public FooInterface{
public:
    virtual void mainFlow() final{
        doFirst();
        mainFlowImplementation();
        doLast();
    }
private:
    void doFirst(){}
    void doLast(){}
};

class FooDerived : public FooBase{
protected:
    void mainFlowImplementation() override { /*some implementation*/}
};
```

R-Value References & Move Semantics

R-Value References

- **C++98 problem**: sending temporary object by value to functions costs depth-copy operations, although the temporary is about to be released.

- Using `std::unique_ptr` is impossible if we do not return it by value and make sure that there is no other copy of it.

R-Value References(Cont...)

- In C++ we can create a reference to an l-value, but we cannot create a reference to an r-value.
- We can create a const reference to it:

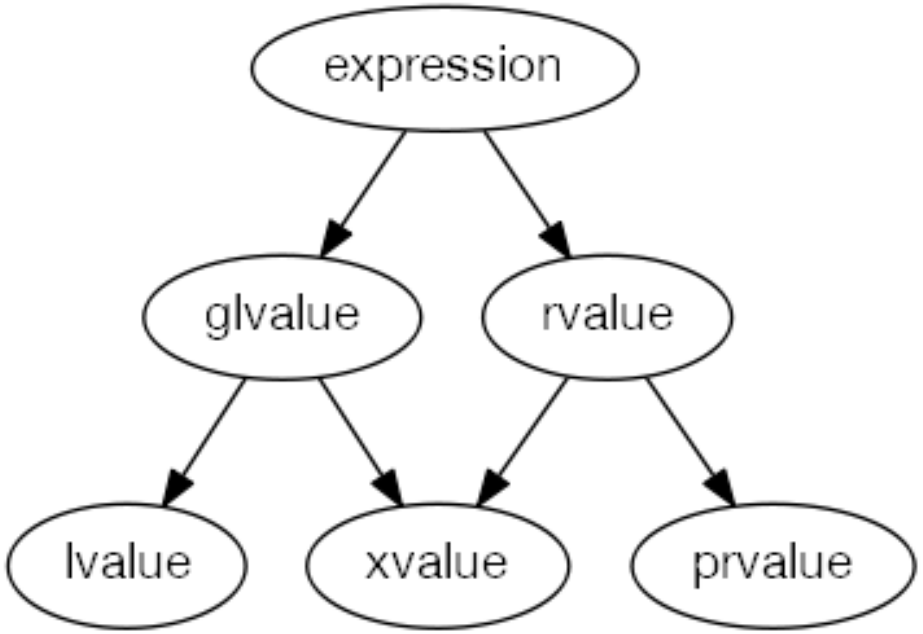
```
int main(){
    int i;

    int& r1 = i;     //OK. reference to l-value
    int& r2 = 20;    //error: cannot declare a reference to rvalue
    const int& r3 = 20; //OK. const reference to rvalue
}
```

R-Value References(Cont...)

- In C++98, an expression is either an **rvalue** or an **lvalue**.

- In C++11, an expression can be an:
  - **rvalue**
  - **lvalue**
  - **xvalue**
  - **glvalue**
  - **prvalue**



Universal References(Cont...)

- **If you can take the address of an expression**, the expression is a **l-value**.
- **If the type of an expression is a l-value reference** (e.g., T& or const T&, etc.), that expression is a **l-value**.
- **Otherwise**, the expression is a **r-value**.
  - Conceptually (and typically also in fact), **r-values** correspond to **temporary objects**, such as those returned from functions or created through implicit type conversions.
  - Most literal values (e.g., 10 and 5.3) are also **r-values**.

R-Value References(Cont...)

```
int x;

int& getRef()
{
    return x;
}

int main(){
    int a;
    a = 1; //a is a lvalue

    getRef() = 4;   //lvalue is not a necessarily a variable
    std::cout<< "x = "<< x << std::endl;
}
```

R-Value References(Cont...)

- An expression is a **r-value** if it results in a temporary object.

```
int x;
int getVal(){
    return x;
}

int main()
{
    getVal(); //the return value is rvalue
}
```

R-Value References(Cont...)

- **Prior to C++11**, if you had a temporary object, you could use a "regular" or "l-value reference" to bind it, but only if it was **const**:

```
const std::string& name = getName(); //ok
std::string& name = getName(); //NOT ok

const std::string& r2 = s1 + s1;
     // okay: l-value reference to const extends lifetime
//r2 += "Test";
     // error: can't modify through reference to const
```

R-Value References(Cont...)

- **C++11 solution**: there's a new kind of reference, an "**r-value reference**", that will let you bind a mutable reference to an r-value, but not an l-value.
- It is declared by **&&** operator.
  - In other words, r-value references are perfect for detecting if a value is temporary object or not.

R-Value References(Cont...)

- **Rvalue references** use the **&&** syntax instead of just **&**, and can be const and non-const, just like **lvalue references**.

```
void printRef(const std::string& str){
    std::cout<<"const ref&: "<<str<<std::endl;
}

void printRef(std::string&& str){
    std::cout<<"ref&&: "<<str<<std::endl;
}

std::string getName(){
    return "Joe";
}

int main(){
    std::string me("Miri");
    printRef(me);
    printRef(getName());
}
```

R-Value References:

- is a references created to a temporary and represents a temporary.
- created with && operator.
- cannot bind a l-value type and cannot point to l-values.

```
int&& rref1 = 10;
int&& rref2 = getVal(); //getVal returns by value
int&& rref3 = 7+2;

//int&& rref4 = a;
     //error: r-value reference cannot bind to a l-value
```

R-Value References(Cont...)

- Between const T& and T&&, a temporary object will bind to T&&.

```
void func(int& x){
    std::cout<<"inside func(int&)\n";
}

void func(int&& x){
    std::cout<<"inside func(int&&)\n";
}

void func(const int& x){
    std::cout<<"inside func(const int&)\n";
}

int main(){
    int x = 3;
    const int y = 4;

    func(x);

    func(y);

    func(3);
}
```

R-Value Refence & Templates

- When using a template function, then it appears that the parameter T&& can capture either l-value and r-value expressions, that it is called "**Universal References**".

Template Type Deduction in C++

- A template function or class only acts as a blueprint.
- The compiler generates code from that blueprint at compile time, after it deduced the type of the function's arguments.
- That occurs when:
  - The function was called.
  - An address to this function was taken.
  - An explicit instantiation was made.
  - A specific specialization was created.

Template Function Resolution

- **Overloading resolution for functions of the same name is done in this order:**
  - Look for an exact match among (non-template) functions.
  - Look for an exact match that can be generated from a function template.
  - Look for the best match, using (in this order):
    - Trivial conversions (exact match).
    - Promotions (short to int).
    - Standard conversions (int to double).
    - User-defined conversions.

Template Type Deduction in C++ (Cont…)

- When the compiler encounters

```
…
f( a, b )
…
```

- Look for all functions named f.
- Creates instances of all templates named f according to parameters.
- Order them by matching quality (1..4).
- **Within** same quality, prefer non-template over template.

Template Type Deduction in C++ (Cont…)

- Suppose we have the following template function:

```
template <class T>
void f(ParamType param);

f(param);
```

- The type T is depending on the type of *param* parameter sent to the function f.

Example 1

```
foo(int)
foo(double)
foo<c>(T)
```

```
#include <iostream>
#include <typeinfo>

void foo(int x) {
    std::cout << "foo(int)\n";
}

void foo(double x) {
    std::cout << "foo(double)\n";
}

template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
```

1

2

3

```
int main()
{
    foo(42);

    foo(42.2);

    foo("abcdef");

    return 0;
}
```

Example 2

```
foo(double)
foo(double)
foo<c>(T)
```

```
#include <iostream>
#include <typeinfo>
//void foo(int x) {
//    std::cout << "foo(int)\n";
//}
void foo(double x) {
    std::cout << "foo(double)\n";
}
template<typename T> void foo(T* x) {
    std::cout << "foo<" << typeid(T).name() << ">(T*)\n";
}
```

2

3

```
int main()
{
    foo(42);

    foo(42.2);

    foo("abcdef");

    return 0;
}
```

Example 3

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
1   std::cout << "foo(int)\n";
}
//void foo(double x) {
2 //   std::cout << "foo(double)\n";
//}
template<typename T> void foo(T x) {
3   std::cout << "foo<" << typeid(T).name() << ">(T)\n";
}
```

```
int main()
{
    foo(42);

    foo(42.2);

    foo("abcdef");

    return 0;
}
```

Example 3 (Cont…)

- Reminder:

Template Function Resolution

Overloading resolution for functions of the same name is done in this order:

- Look for an exact match among (non-template) functions.
- Look for an exact match that can be generated from a function template.
- Look for the best match, using (in this order):
  - Trivial conversions (exact match).
  - Promotions (short to int).
  - Standard conversions (int to double).
  - User-defined conversions.

Example 3 (Cont…)

```
foo(int)
foo<d>(T)
foo<PKc>(T)
```

```
#include <iostream>
#include <typeinfo>
void foo(int x) {
1   std::cout << "foo(int)\n";
}
//void foo(double x) {
2 //   std::cout << "foo(double)\n";
//}
template<typename T> void foo(T x) {
3   std::cout << "foo<" << typeid(T).name() << ">(T)\n";
}
```

```
int main()
{
    foo(42);

    foo(42.2);

    foo("abcdef");

    return 0;
}
```

Example 4

```
template<typename T>
void f(T x, T y) {
      std::cout << "Template" << std::endl;
}
```

```
void f(int w, int z) {
      std::cout << "Non-template" << std::endl;
}
```

```
int main(){
   f( 1 ,  2 );
   f('a', 'b');
   f( 1 , 'b');
}
```

```
Non-template
Template
Non-template
```

Example 5

```
template <typename T> void f(T) { cout << "Less specialized");}   1
template <typename T> void f(T*) { cout<< "More specialized");}   2
int main() {
    int i =0;
    int *pi = &i;
    f(i);   // Calls less specialized function.
    f(pi); // Calls more specialized function.
}
```

Is there a type that fits 1 and will not fit 2?

If so, 2 is more specialized and should be preferred.

Reference Collapsing Rules

- It is permitted to form references to references through type manipulations in templates or typedefs, in which case the reference collapsing rules apply: **rvalue reference to rvalue reference collapses to rvalue reference, all other combinations form lvalue reference**:
  - A& & becomes A&
  - A& && becomes A&
  - A&& & becomes A&
  - A&& && becomes A&&

Reference Collapsing Rules (Cont…)

```
typedef int&  lref;
typedef int&& rref;

int n;
lref&  r1 = n; // type of r1 is int&

lref&& r2 = n; // type of r2 is int&: A& && becomes A&

rref&  r3 = n; // type of r3 is int&: A&& & becomes A&

rref&& r4 = 1; // type of r4 is int&&: A&& && becomes A&&
```

Case 1: Passing *param* by reference/pointer

- If the passed argument is reference – ignore it.
- Do pattern matching to deduce T.

```
template <class T>
void f(T& param);
```

param's type	T type	
`int&`	`int`	`int x=22;` `f(x);`
`const int&`	`const int`	`const int cx = x;` `f(cx);`
`const int&`	`const int`	`const int& rx = x;` `f(rx);`

Template Type Deduction in C++ (Cont...)

```
template <class T>
void f(const T& param);
```

param's type	T type	
const int&	int	int x=22; f(x);
const int&	int	const int cx = x; f(cx);
const int&	int	const int& rx = x; f(rx);

Template Type Deduction in C++ (Cont...)

- The same rules apply on “auto” reduction:

auto type	v’s type	
`int`	`int&`	`int x=22;` `auto& v1 = x;`
`const int`	`const int&`	`const int cx = x;` `auto& v2 = cx;`
`const int`	`const int&`	`const int&  rx = x;` `auto& v3 = rx;`

Case 2: Universal References

- **T&&** can capture either **l-value** or **r-value** (templates only!!!).

```
template <class T>
void f(T&& param);

f(expr);
```

- It depends if “expr” is l-value or r-value.

- **If** *expr* is l-value with deduced type E,
  T deduced as **E&**.

Template Type Deduction in C++ (Cont...)

- **Reference Collapsing Rules again:**
- A& & becomes A&
- A& && becomes A&
- A&& & becomes A&
- A&& && becomes A&&

Template Type Deduction in C++ (Cont...)

what T&& represents depends on what it is initialized with!

```
template <class T>
void f(T&& param);
```

param's type	T type	
int&	int&	int x=22; f(x); //x is l-value
const int&	const int&	const int cx = x; f(cx); //cx is l-value
const int&	const int&	const int& rx = x; f(rx); //rx is l-value
int&&	int	f(22); //r-value

Case 3: By value

- If *expr* is &, then the reference is ignored.
- If *expr* is const or volatile, it is ignored.
- Deduce T.

```
template <class T>
void f(T param);
```

param's type	T type	
int	int	int x=22; f(x);
int	int	const int cx = x; f(cx);
int	int	const int& rx = x; f(rx);

Template Type Deduction in C++ (Cont...)

CppCon 2014: Scott Meyers "Type Deduction and Why You Care"

- using auto also creates a brand-new variable.

```
int x = 22;
const int cx = x;
const int& rx = x;

auto v1 = x;     //v1's type is int (auto is int)
auto v2 = cx;    //v2's type is int (auto is int)
auto v3 = rx;    //v3's type is int (auto is int)

auto& v4 = rx;   //v4's type is const int&
auto&& v5 = rx;  //v5's type is const int& (rx is lvalue)
```

Move Semantics

- The move semantics allow **modifying rvalues** (previously considered immutable and indistinguishable from const T& types).

Move Semantics (cont...)

- Suppose we have a CMyVector class:

```
class CMyVector {
private:
    int* m_pArr;
    //...
};
```

- For this class we'll need to implement copy constructor, destructor and assignment operator.

Move Semantics (cont...)

```
class CMyVector {
public:
    //...
    CMyVector& operator=(const CMyVector& vec) {
        // [...]
        // Make a clone of what vec.m_pArr refers to.
        // Destruct the resource that m_pArr refers to.
        // Attach the clone to m_pArr.
        // [...]

        return *this;
    }
private:
    int* m_pArr;
};
```

Move Semantics (cont...)

- Similar reasoning applies to the copy constructor.
- Now suppose CMyVector is used as follows:

```
CMyVector foo() {
    return CMyVector();
}

int main() {
    CMyVector x;
    // perhaps use x in various ways
    x = foo();
    //...
}
```

Move Semantics (cont…)

- The last line above
  - **clones** the resource from the temporary returned by foo,
  - **destructs the resource** held by x and replaces it with the clone,
  - **destructs the temporary** and thereby releases its resource.

Move constructor and move assignment operator (Cont...)

```
#include <iostream>

class MyString
{
public:
  MyString(const char* str=nullptr);
  ~MyString();
private:
  char* m_str;
};
```

Move constructor and move assignment operator (Cont…)

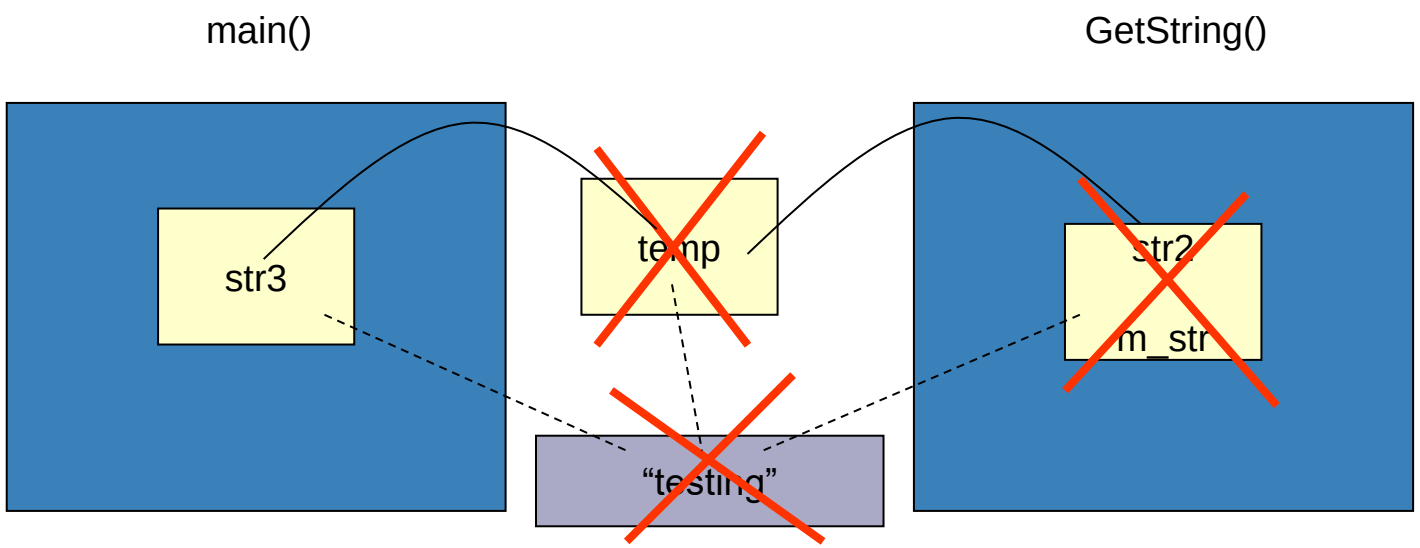
```
MyString::MyString(const char* str1/*=nullptr*/)
{
        std::cout<<"creating a string\n";
        if(str1){
                m_str = new char[strlen(str1)+1];
                strcpy(m_str,str1);
        }
        else{
                m_str = nullptr;
        }
}
```

```
MyString::~MyString()
{
        if(m_str){
                delete[] m_str;
                m_str = nullptr;
        }
        std::cout<<"deleting a string\n";
}
```

Move constructor and move assignment operator (Cont...)

```
MyString GetString()
{
        MyString str2("testing");
        return str2;
}

int main()
{
        MyString str3( GetString() );
}
```



Move Semantics (cont...)

- Rather obviously, it would be ok, and much more efficient, to **swap resource pointers (handles) between x and the temporary**, and then let the temporary's destructor destruct x's original resource.

- In other words, in the special case where the right hand side of the assignment is a r-value, we want the copy assignment operator to act like this:

```
// [...]
// swap m_pArr and x.m_pArr
// [...]
```

Move Semantics (Cont...)

- **Move semantics** allows you to avoid unnecessary copies when working with temporary objects that are about to evaporate, and whose resources can safely be taken from that temporary object and used by another.

Example 1: l-value and r-value

```
class MyClass
{
public:
        MyClass() = default;
        MyClass(const MyClass& m) {std::cout<<"copy-ctor\n";}
        MyClass(MyClass&& m) {std::cout<<"move-ctor\n";}
};

MyClass get(MyClass a){
        return a;
}

void foo(MyClass m){
        std::cout<<"inside by-value foo\n";
}

int main(){
        MyClass obj;

        foo(obj);

        foo(get(obj));

        return 0;
}
```

- **דוגמא 1: העברת אובייקט לפונקציה by value.**
  - אם האובייקט שנשלח הוא l-value אזי:
    נבנה את הפרמטר עם copy ctor.
  - אם האובייקט שנשלח הוא r-value אזי:
    נבנה את הפרמטר עם move ctor.

Example 2: l-value and r-value

```
class MyClass
{
public:
        MyClass() = default;
        MyClass(const MyClass& m) {std::cout<<"copy-ctor\n";}
        MyClass(MyClass&& m) {std::cout<<"move-ctor\n";}
};

void foo(MyClass& m){
        std::cout<<"inside lvalue foo\n";
}

void foo(MyClass&& m){
        std::cout<<"inside rvalue foo\n";
}

MyClass get(MyClass a){
        return a;
}

int main(){
        MyClass obj;

        foo(obj);

        foo(get(obj));
}
```

Example 3: l-value and r-value

```
class MyClass
{
public:
        MyClass()  = default;
        MyClass(const MyClass& m) {std::cout<<"copy-ctor\n";}
        MyClass(MyClass&& m) {std::cout<<"move-ctor\n";}
};

/*
void foo(MyClass& m)
{
        std::cout<<"inside lvalue foo\n";
}
*/

void foo(MyClass&& m)
{
        std::cout<<"inside rvalue foo\n";
}

MyClass get(MyClass a){
        return a;
}

int main(){
        MyClass obj;

        foo(obj);

        foo(get(obj));
}
```

- **דוגמא 3: הפונקציה מקבלת את הפרמטר כ.&&-**
  - אם שלחנו לפונקציה l-value, אזי:
    שגיאת קומפילציה !!
    &&לא יכול לקבל l-value!
  - אם שלחנו לפונקציה r-value, אזי:
    נבנה את הפרמטר של הפונקציה עם ה.move ctor-

Example 4: l-value and r-value

```
class MyClass
{
public:
        MyClass() = default;
        MyClass(const MyClass& m) {std::cout<<"copy-ctor\n";}
        MyClass(MyClass&& m) {std::cout<<"move-ctor\n";}
};


void foo(/*const*/ MyClass& m){      //will work with the const for rvalue
        std::cout<<"inside lvalue foo\n";
}

/*
void foo(MyClass&& m){
        std::cout<<"inside rvalue foo\n";
}
*/

MyClass get(MyClass a){
        return a;
}

int main(){
        MyClass obj(3);

        foo(obj);

        foo(get(obj));
}
```

- **דוגמא 4: הפונקציה מקבלת את הפרמטר by reference.**
  - אם שלחנו לפונקציה l-value, אזי:
    לא נקרא שום בנאי, כי העברנו by reference.
  - אם שלחנו לפונקציה r-value, אזי:
    אם הפרמטר לא – const שגיאת קומפילציה.
    אחרת -נבנה את הפרמטר של הפונקציה עם ה move-ctor.

Move constructor and move assignment operator (Cont...)

```
template <typename T>
class Buffer
{
    std::string          _name;
    size_t               _size;
    std::unique_ptr<T[]> _buffer;

public:
    // default constructor
    Buffer() : _size(16), _buffer(new T[16])
    {}

    // constructor
    Buffer(const std::string& name, size_t size) :
        _name(name), _size(size), _buffer(new T[size])
    {}

    // copy constructor
    Buffer(const Buffer& copy) :
        _name(copy._name), _size(copy._size), _buffer(new T[copy._size])
    {
        T* source = copy._buffer.get();
        T* dest = _buffer.get();
        std::copy(source, source + copy._size, dest);
    }
```

Move constructor and move assignment operator (Cont...)

```
// copy assignment operator
Buffer& operator=(const Buffer& copy)
{
    if (this != &copy)
    {
        _name = copy._name;

        if (_size != copy._size)
        {
            _buffer = nullptr;
            _size = copy._size;
            _buffer = _size > 0 > new T[_size] : nullptr;
        }

        T* source = copy._buffer.get();
        T* dest = _buffer.get();
        std::copy(source, source + copy._size, dest);
    }

    return *this;
}
```

Move constructor and move assignment operator (Cont...)

```
    // move constructor
    Buffer(Buffer&& temp) :
        _name(std::move(temp._name)), _size(temp._size),
        _buffer(std::move(temp._buffer))
    {
        temp._buffer = nullptr;
        temp._size = 0;
    }

    // move assignment operator
    Buffer& operator=(Buffer&& temp)
    {
        assert(this != &temp); // assert if this is not a temporary

        _buffer = nullptr;
        _size = temp._size;
        _buffer = std::move(temp._buffer);

        _name = std::move(temp._name);

        temp._buffer = nullptr;
        temp._size = 0;

        return *this;
    }
};
```

```
template <typename T>
Buffer<T> getBuffer(const std::string& name)
{
    Buffer<T> b(name, 128);
    return b;
}
int main()
{
    Buffer<int> b1;
    Buffer<int> b2("buf2", 64);
    Buffer<int> b3 = b2;
    Buffer<int> b4 = getBuffer<int>("buf4");
    b1 = getBuffer<int>("buf5");
    return 0;
}
```

Move constructor and move assignment operator (Cont...)

- If no user-defined move constructors are provided for a class type (struct, class, or union), and all of the following is true:
  - there are no user-declared copy constructors;
  - there are no user-declared copy assignment operators;
  - there are no user-declared move assignment operators;
  - there are no user-declared destructors;

  then the compiler will declare a move constructor as a non-explicit inline public member of its class with the signature T::T(T&&).

```
class CMyClassA {
public:
    CMyClassA() = default;
    CMyClassA(const CMyClassA& cobj) { std::cout << "copy ctor\n"; }
    CMyClassA(CMyClassA&& cobj) { std::cout << "move ctor\n"; }
};

void funcByValue(CMyClassA obj) {
//  std::cout << "funcByValue\n";
}

void funcByRef(CMyClassA& obj) {
//  std::cout << "funcByRef\n";
}

void funcByRefRef(CMyClassA&& obj) {
//  std::cout << "funcByRefRef\n";
}

template<class T>
void funcTempByRefRef(T&& obj) {

}
```

```
void check() {
    CMyClassA obj; //lvalue object.

    std::cout << "send to funcByValue:\n";
    funcByValue(obj);   //expect to call copy ctor
    funcByValue(createObj());   //expect to call move ctor
    std::cout << "\n--------------\n";

    std::cout << "send to funcByRef:\n";
    funcByRef(obj); //expect to not call any ctor
    funcByRef(createObj()); //expect to call move ctor
    std::cout << "\n--------------\n";

    std::cout << "send to funcByRefRef:\n";
//  funcByRefRef(obj);  //compilation error: cannot bind l-value to r-value reference
    funcByRefRef(createObj());  //expect to call move ctor
    std::cout << "\n--------------\n";

    std::cout << "send to funcTempByRefRef:\n";
    funcTempByRefRef(obj);  //l-value is deduced to be T& -> no ctor is called
    funcTempByRefRef(createObj());  //r-value is deduced to T&& -> move ctor
    std::cout << "\n--------------\n";

}
```

Perfect Forwarding

- What will happen if from a template function that gets a universal reference, we want to pass the argument on to another function?

- Is it an l-value or an r-value?

perfect forwarding (cont...)

```
class CMyClass {
public:
    CMyClass() = default;
    CMyClass(const CMyClass& cobj) { std::cout << "copy ctor\t"; }
    CMyClass(CMyClass&& cobj) { std::cout << "move ctor\t"; }
};

CMyClass create() {
    return CMyClass();
}
```

perfect forwarding (cont...)

- Case 1: sending l-value to the template function:

```
template <class T>
void relayRV(T&& cobj) {
    std::cout << "in relay\t";
    foo(cobj); //send by value
}

int main() {
    CMyClass cobj;

    //pass a l-value object
    relayRV(cobj);  //l-value is passed as T&
}
```

perfect forwarding (cont...)

- Case 2: sending r-value to the template function:

```
template <class T>
void relayRV(T&& cobj) {
    std::cout << "in relay\t";
    foo(cobj); //send by value
}

int main() {
    //pass a r-value object
    relayRV(create()); //r-value is passed with move ctor
}
```

std:move

- A casting expression from any type to an r-value. Compilation-time!

- It casts the l-value expression x of type X to an r-value expression of type X.
- move can also accept an rvalue.

std::forward

- A conditional casting from any type to an r-value. Compilation time!
  - If the expression is an l-value, that it won’t be cast.
  - If the expression is an r-value, that it will be casted to r-value.

Perfect forwarding (cont...)

- If a template function gets a T object and wants to forward it to another function, we would like it to be done as efficient as possible.

- If the parameter is passed as an r-value, than we want it to also be passed as r-value, and not creating a new object from it.

- **It will be done by declaring the type of argument as T&& and use the std::forward casting.**

Perfect forwarding (cont…)

```
class CMyClass {
public:
    CMyClass() = default;
    CMyClass(const CMyClass& cobj) { std::cout << "copy ctor\t"; }
    CMyClass(CMyClass&& cobj) { std::cout << "move ctor\t"; }
};

CMyClass create() {
    return CMyClass();
}

void foo(CMyClass cobj) {
    std::cout << "in foo\n";
}

template <class T>
void relayRV(T&& cobj) {
    foo(std::forward<T>(cobj));
}
```

Perfect forwarding (cont…)

```
int main() {
    CMyClass cobj;
    std::cout << "--------------\n";

    //pass a l-value object
    relayRV(cobj);
    std::cout << "--------------\n";

    //pass a r-value object
    relayRV(create());
    std::cout << "--------------\n";

    return 0;
}
```

Copy Elision Guarantee

Copy Elision Guarantee

- A technique of the compiler to eliminate the temporary objects.
- It omits copy and move constructors, resulting in zero-copy pass-by-value semantics.
- Most compilers implements it anyway, but since C++17 it is part of the standard.

```
//compile with flag -fno-elide-constructors
class MyClass{
public:
    MyClass() {std::cout<<"[MyClass] default ctor\n";}
    MyClass(const MyClass&) {std::cout<<"[MyClass] copy ctor\n";}
    MyClass(MyClass&&) {std::cout<<"[MyClass] move ctor\n";}
    ~MyClass() {std::cout<<"[MyClass] dtor\n";}
};

MyClass getObject(){
    std::cout<<"[getObject] inside function\n";
    return MyClass();
}

int main(){
    std::cout<<"[main] stating main\n";
    MyClass m = getObject();
    std::cout<<"[main] ending main\n";
}
```

Copy Elision Guarantee (Cont…)

- Compiling with the flag `-fno-elide-constructors` :

```
[main] stating main
[getObject] inside function
[MyClass] default ctor
[MyClass] move ctor
[MyClass] dtor
[MyClass] move ctor
[MyClass] dtor
[main] ending main
[MyClass] dtor
```

Copy Elision Guarantee (Cont…)

- Compiling without the flag:

```
[main] stating main
[getObject] inside function
[MyClass] default ctor
[main] ending main
[MyClass] dtor
```

Copy Elision Guarantee (Cont…)

- This rule claims that the following statements are **always** identical.

```
auto x = Foo{};
Foo x{};
```

- We can now use the left-to-right initialization style without worrying about performance and we can also use it for non-movable/non-copyable types.

```
//compile with flag -fno-elide-constructors
class MyClass{
public:
    MyClass() {std::cout<<"[MyClass] default ctor\n";}
    MyClass(int x) {std::cout<<"[MyClass] conversion(int) ctor\n";}
    MyClass(const MyClass&) {std::cout<<"[MyClass] copy ctor\n";}
    MyClass(MyClass&&) {std::cout<<"[MyClass] move ctor\n";}
    ~MyClass() {std::cout<<"[MyClass] dtor\n";}
};

int main(){
    std::cout<<"[main] stating main\n";

    MyClass m = 3;

    std::cout<<"[main] ending main\n";
}
```

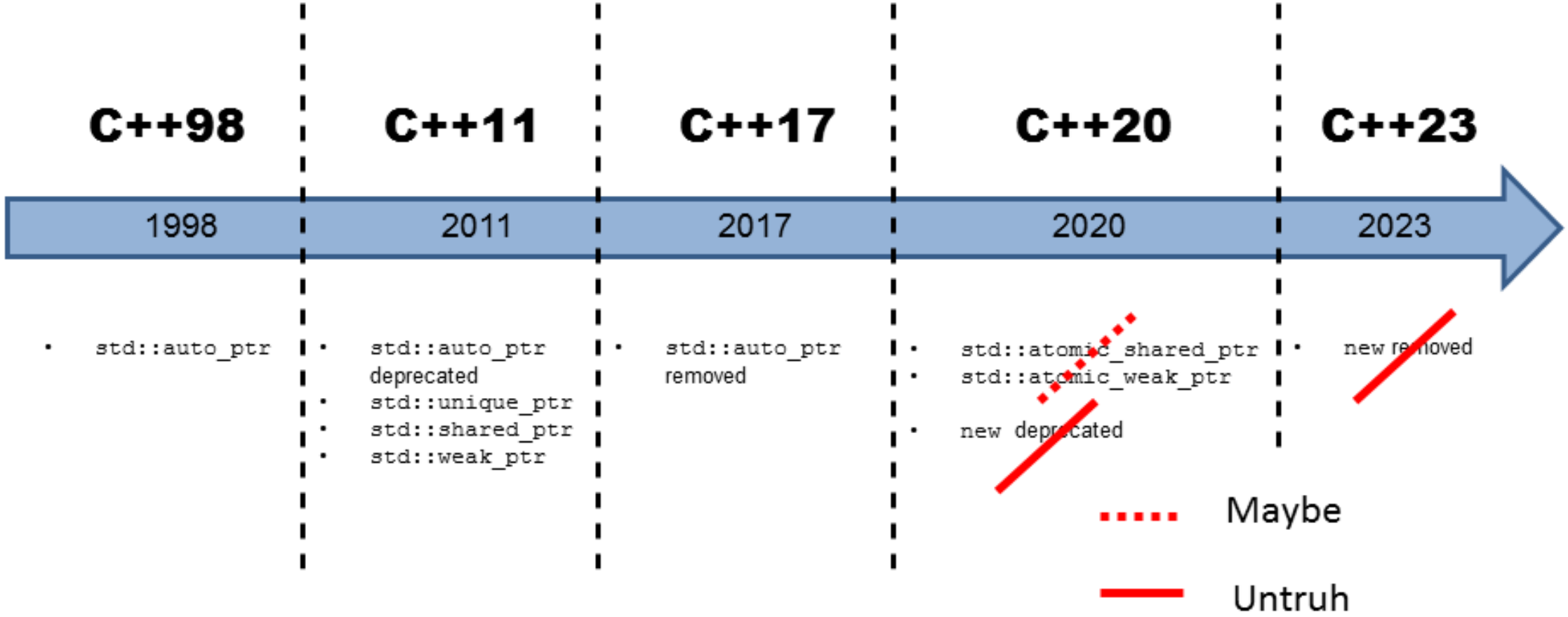
Smart Pointers

RAII – Resource Acquisition Is Initialization

- RAII is a C++ programming technique which binds the life cycle of a resource that must be acquired before use to the lifetime of an object.
  - For example: allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply.

- It guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition.
  - Especially important when an exception is raised and only stack-based objects are cleaned.

Evolution of pointers in C++



RAII Idiom

- **R**esource **A**cquisition **I**s **I**nitialization or **RAII**, is a C++ programming technique which binds the life cycle of a resource that must be acquired before use to the lifetime of an object.
  - Example: Objects that handles allocated heap memory, thread of execution, open socket, open file, locked mutex, disk space, database connection—anything that exists in limited supply.
- RAII guarantees that the resource is available to any function that may access the object.
- It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition.

RAII Idion (Cont…)

- RAII in practice:
  - Encapsulate each resource into a class, where:
    - **The constructor** acquires the resource and establishes all class invariants or throws an exception if that cannot be done,
    - **The destructor** releases the resource and never throws exceptions;
  - Always use the resource via an instance of a RAII-class that either
    - Has automatic storage duration or temporary lifetime itself, or
    - Has lifetime that is bounded by the lifetime of an automatic or temporary object

- Move semantics make it possible to safely transfer resource ownership between objects, across scopes, and in and out of threads, while maintaining resource safety.

RAII Idion (Cont…)

- Use RAII to prevent leaks:

```
void f(int i)   // Bad: possible leak
{
    int* p = new int[12];
    // ...
    if (i < 17) throw Bad{"in f()", i};
    // ...
}
```

```
void f(int i)   // OK: resource management done by local object
{
    vector<int> v(12);
    // ...
    helper(i);   // might throw
    // ...
}
```

MKBN

Smart Pointers

- Smart Pointers are objects that manage the lifetime of the resources they wrap (RAII).
  - Smart Pointer is a pattern in which the raw pointer is wrapped with an object that raise on the stack, thus its destructor is called when the scope is ended, releasing the wrapped resource.
- This is the C++ answer to garbage collection.
- Uses reference counting.
- Uses the fact that the compiler handles local objects.

Smart Pointers (Cont…)

- A smart pointer looks and behaves like a regular C++ pointer by overloading `*`, `->`, `[ ]` etc'.
- There are three types of smart pointers: **Unique Pointer**, **Shared Pointer** and **Weak Pointer.**
  - **`std::unique_ptr`** – takes ownership of a pointer. Thus, it cannot be copied.
  - **`std::shared_ptr`** – allow shared objects to have multiple owners.
  - **`std::weak_ptr`** – similar to `shared_ptr` but doesn't affect the reference count. Thus, it cannot be just dereferenced.

Problems with raw pointers

- Does this pointer point to an object or to an array?
- Do I need to destroy the object it points to, or I'm not the owner of it?
- If I need to destroy the object, how should I do it? delete? some dedicated destruction function?
- If I need to use delete, should I use delete[] in this case?
- How can I assure that the destruction is done only once?
- Is that pointer valid or dangling?

Smart Pointer Implementation

- We'll suggest a simple implementation of `unique_ptr` and `shared_ptr`.
- When designing the class, we need to think what it its API first.
  - It is a good idea to start with a usage test and then implement the API.

Members | Constructors | Destructor | Operators

Smart Pointer Implementation (Cont…)

```
template <class T>
class my_ptr
{
public:
    //constructors

    //destructor
    ~my_ptr() {
    }

    //assignment operator

    //operator ->

    //operator *
private:

};
```

Smart Pointer Implementation (Cont…)

- In the **unique pointer** implementation, we don't want to allow copying of one object to another because that would mean the address of the underlying memory being copied that will leave the ownership of that memory with more than one objects.

```
unique_ptr(const unique_ptr& obj) = delete;
    // copy constructor is deleted

unique_ptr& operator=(const unique_ptr& obj) = delete;
    // copy assignment is deleted
```

```
unique_ptr(unique_ptr&& dyingObj) // move constructor
{
    // Transfer ownership of memory pointed by dyingObj
    this->ptr = dyingObj.ptr;
    dyingObj.ptr = nullptr;
}
void operator=(unique_ptr&& dyingObj) // move assignment
{
    // cleanup any existing data
    // Transfer ownership of memory pointed
    this->ptr = dyingObj.ptr;
    dyingObj.ptr = nullptr;
}
```

https://medium.com/swlh/c-smart-pointers-and-how-to-write-your-own-c0adcbdce04f

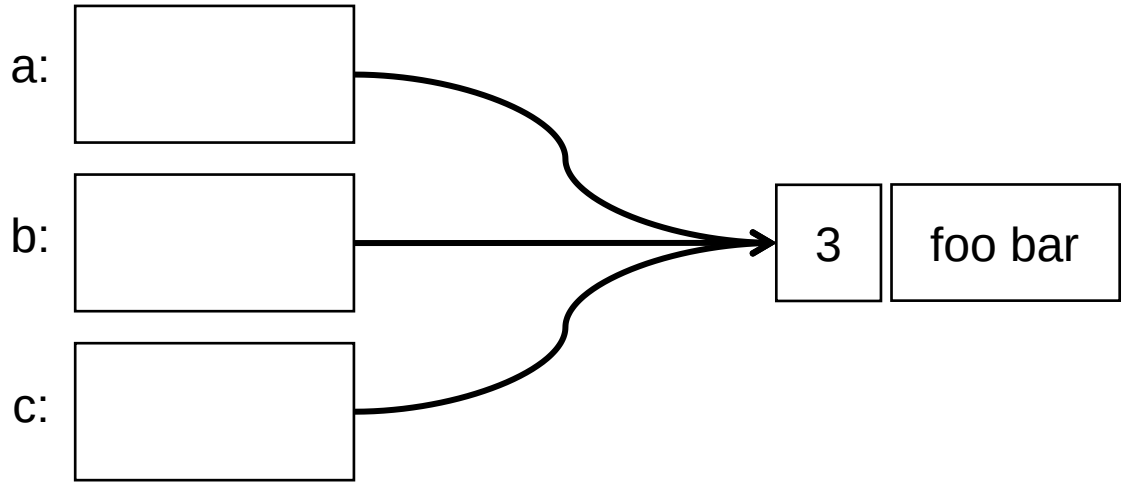
Smart Pointer Pattern (Cont…)

- In the **shared pointer** implementation, a reference count is added to make sure that the resource is not released until there is no one who holds this resource anymore.

```
template<class T>
class shared_ptr{
private:
    T * _ptr = nullptr;
    size_t* _refCount = nullptr;
public:
    shared_ptr() : _ptr(nullptr), _refCount(new size_t(0)){}
};
```

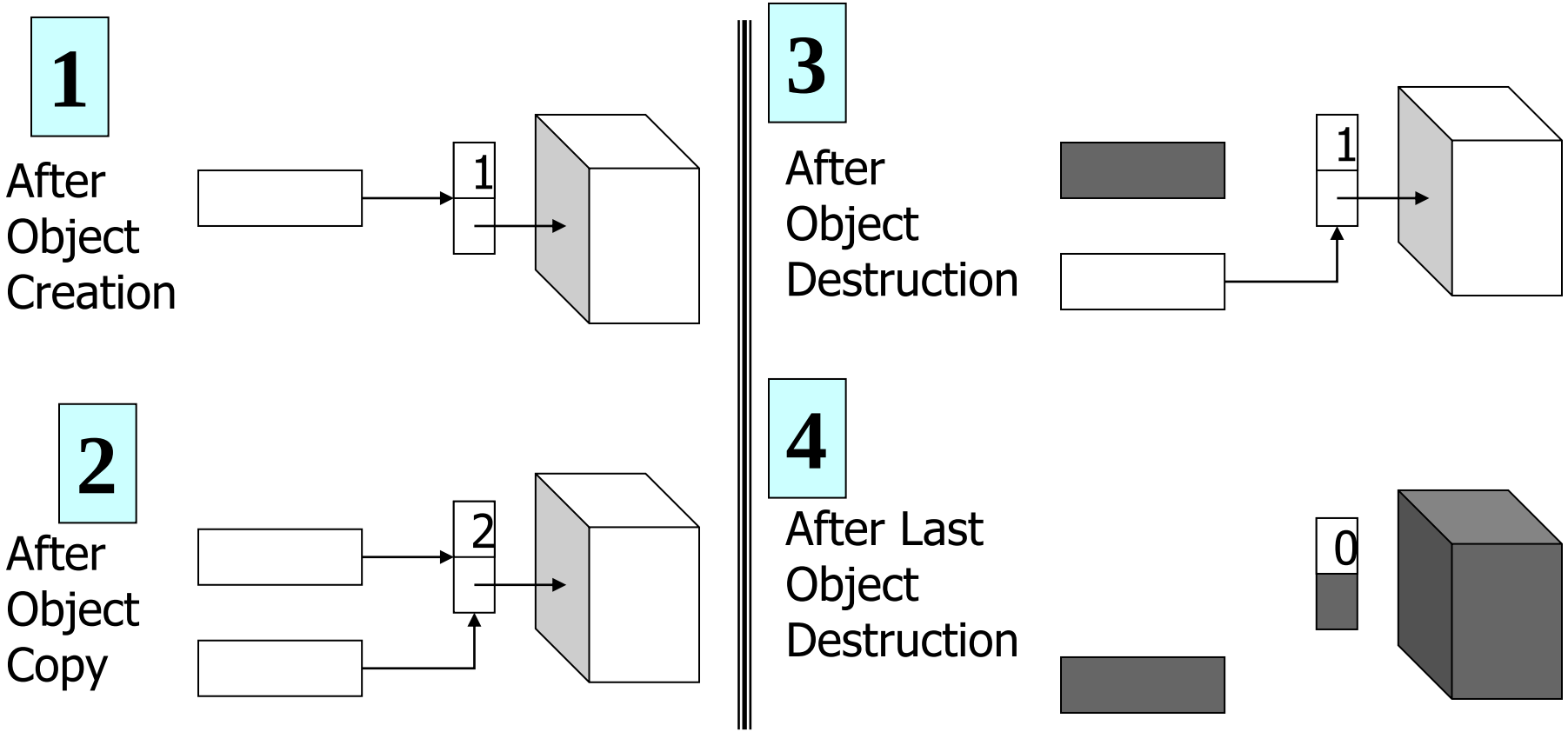
Reference Counting

- Store a **reference counter** that counts number of pointers

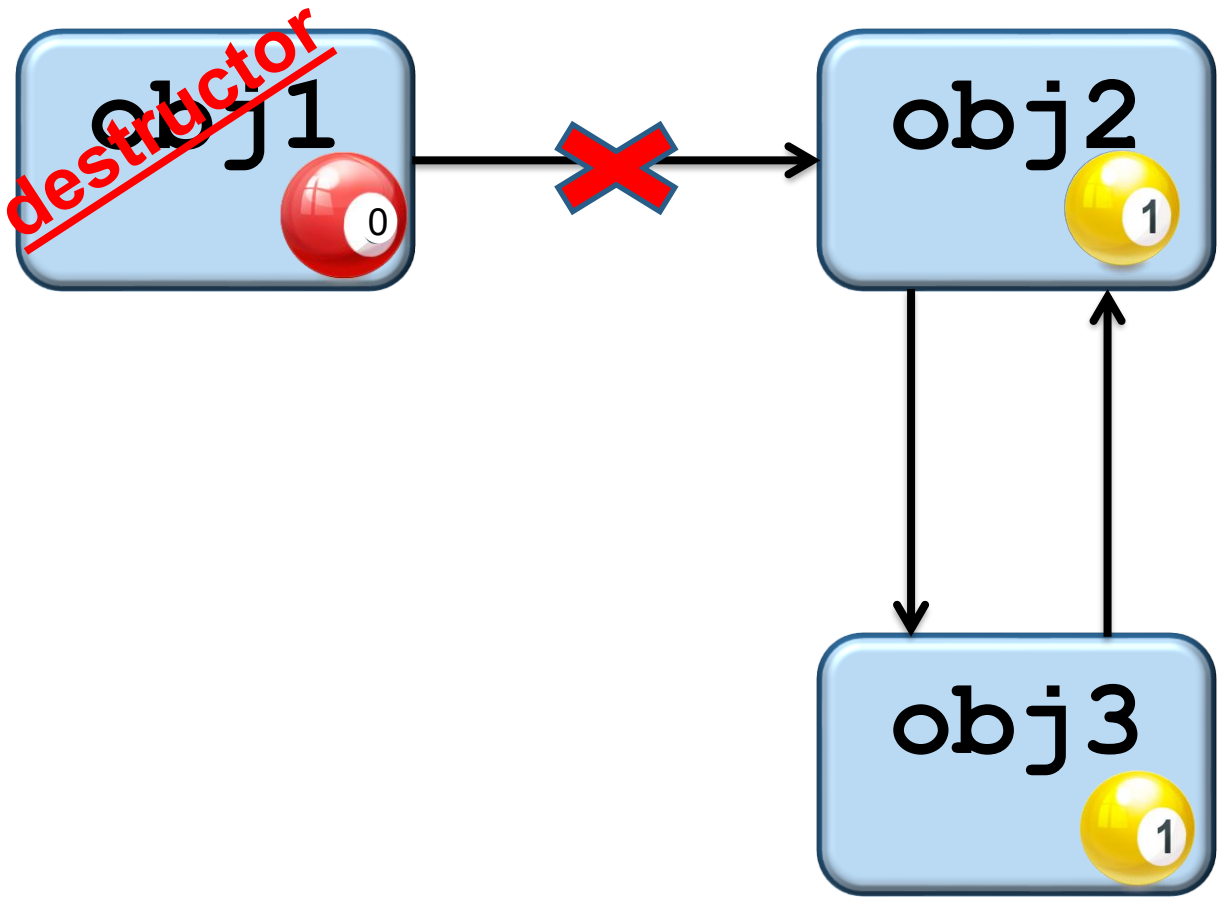


- Reference counter shows whether the object is in use

Reference Counting



Reference counting and cycles



MKBN

```
shared_ptr(const shared_ptr & obj){
    _ptr = obj.ptr;        // share the underlying pointer
    _refCount = obj.refCount; // share refCount
    if (nullptr != obj.ptr){
        // if the pointer is not null, increment the refCount
        (*_refCount)++;
    }
}

shared_ptr& operator=(const shared_ptr& obj){
    if(this != &obj){
        _ptr = obj.ptr; // share the underlying pointer
        _refCount = obj.refCount; // share refCount
        if (nullptr != obj.ptr){
            // if the pointer is not null, increment the refCount
            (*_refCount)++;
        }
    }
    return *this;
}
```

C++11 Smart Pointers

- **C++11** added support for smart pointers in the std namespace in the `<memory>` header file:
  - `unique_ptr`: should be used when ownership of a memory resource does not have to be shared (it doesn't have a copy constructor), but it can be transferred to another `unique_ptr` (move constructor exists).
  - `shared_ptr`: should be used when ownership of a memory resource should be shared (hence the name).
  - `weak_ptr`: holds a reference to an object managed by a `shared_ptr` but does not contribute to the reference count; it is used to break dependency cycles.

std::unique_ptr

```
void foo(int* p){
    std::cout<< *p <<std::endl;
}

int main(){
    std::unique_ptr<int> p1(new int(42));
    // we do not have copy constructor for unique_ptr:
    // std::unique_ptr<int> p2 = p1;    //ERROR!
    std::unique_ptr<int> p2 = std::move(p1);//call move ctor of unique_ptr
                                            //and transfer ownership
    //p1 is empty so we will not enter the if block
    if(p1){
        foo(p1.get());
    }
    (*p2)++;
    if(p2){
        foo(p2.get());
    }
}
```

std::shared_ptr

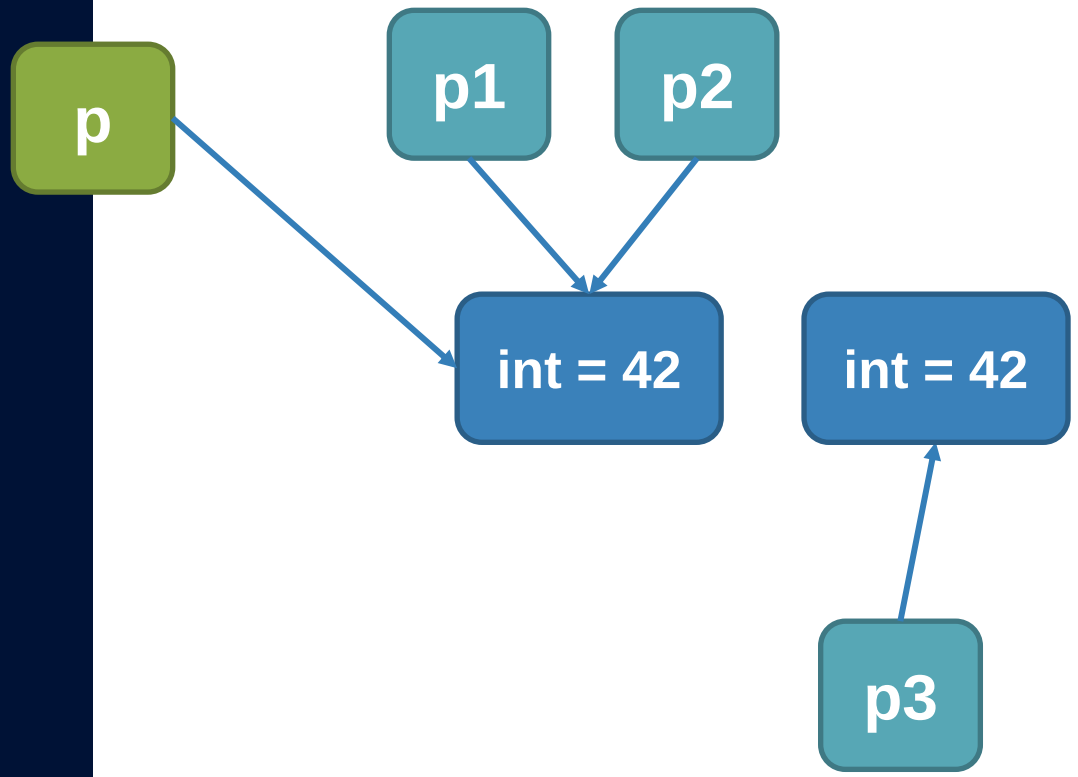
```
void foo(int* p){
}

void bar(std::shared_ptr<int> p){
    ++(*p);
}

int main(){
    std::shared_ptr<int> p1(new int(42));
    std::shared_ptr<int> p2 = p1;

    //same as sending new int(42)
    auto p3 = std::make_shared<int>(42);

    bar(p1);
    foo(p2.get());
}
```



std::shared_ptr (Cont…)

- The existence of the reference count has performance implications:
  - **`std::shared_ptr`**s are twice the size of a raw pointer.
  - The memory for the reference count must be dynamically allocated.
  - Increments and decrements of the reference count must be atomic.

std::shared_ptr(Cont…)

- `make_shared<T>` is a non-member function and has the advantage of allocating memory for the shared object and the smart pointer with a single allocation, as opposed to the explicit construction of a `shared_ptr` via the constructor, that requires at least two allocations.

- In addition to possible overhead, there can be situations where memory leaks can occur because of that.

std::shared_ptr (Cont…)

- Using the constructor of `shared_ptr` can cause problems:

```
(*pn)++;
std::cout<<"pn="<<*pn<<", spn="<<*spn<<std::endl;
std::cout<<spn.use_count()<<std::endl;

delete pn;
std::cout<<spn.use_count()<<std::endl; //still 1!
std::cout<<*spn;    //undefined behavior!
```

std::shared_ptr(Cont…)

- In the next example memory leaks could occur if seed() throws an error.

```
void foo(std::shared_ptr<int> p, int init){
    *p = init;
}

int main(){
    foo(std::shared_ptr<int>(new int(42)), seed());

    foo(std::make_shared<int>(42),seed());

}
```

std::weak_ptr

- `std::weak_ptr` is used when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else.
- It is used to track the object, and it is **converted** to `std::shared_ptr` to assume temporary ownership.
- If the original `std::shared_ptr` is destroyed at this time, the object's lifetime is extended until the temporary `std::shared_ptr` is destroyed as well.
  - You get a `shared_ptr` to the referred object by calling `lock()`, in order to access the object.

std::weak_ptr(Cont...)

- If you try to lock on an expired `weak_ptr` (the object is weakly reference has been released) you get an empty `shared_ptr`.

```
int main(){
    auto p1 = std::make_shared<int>(42);
    std::weak_ptr<int> wp = p1;

    {
        auto sharedP = wp.lock();   //a shared_ptr is created
        std::cout<<*sharedP<<std::endl;
    }

    p1.reset();

    if(wp.expired()){
        std::cout<< "expired" <<std::endl;
    }
}
```

Custom Deleter

- By default, the destruction of the smart pointer is done via the delete method.
- What if we would like to use a different custom function?
  - For example: add a log entry first, or in embedded systems call the delete from the host code that will be performed on the device memory context, etc’.
- During construction we can configure the smart pointer with an arbitrary function (as well as function object, lambda expression) to be invoked when it’s time for their resources to be destroyed.

Custom Deleter (Cont…)

```
#include <iostream>
#include <memory>

struct MyObject{
    MyObject() {std::cout<<"creating the object\n";}
    ~MyObject() {std::cout<<"destructing the object\n";}
};

auto get_object(){
    auto deleter = [](MyObject* p){
        std::cout<<"in the custom deleter code\n";
        delete p;
    };
    std::unique_ptr<MyObject,decltype(deleter)> pObj(new MyObject,deleter);
    return pObj;
}

int main(){
    auto pObj = get_object();
}
```

Custom Deleter (Cont…)

- In std::shared_ptr the custom delete function is not part of the pointer's type.

```
auto get_object(){
    auto deleter = [](MyObject* p){
        std::cout<<"in the custom deleter code\n";
        delete p;
    };
    std::shared_ptr<MyObject> pObj(new MyObject,deleter);
    return pObj;
}
int main(){
    auto pObj = get_object();
}
```

Custom Deleter (Cont…)

- Notice that the deleter function object requires extra space from the shared object's size. This is why you prefer the lambda notation (or function pointer).

Custom Deleter (Cont…)

```
void deleter_function(MyObject*) {}

struct DeleterFunctionObject {
    void operator()(MyObject*) {}
    size_t some_data[10];
};

auto get_object() {
    auto deleter = [](MyObject* p) {
        std::cout << "in the custom deleter code\n";
        delete p;
    };
    return std::unique_ptr<MyObject, decltype(deleter)>{new MyObject, deleter};
}
```

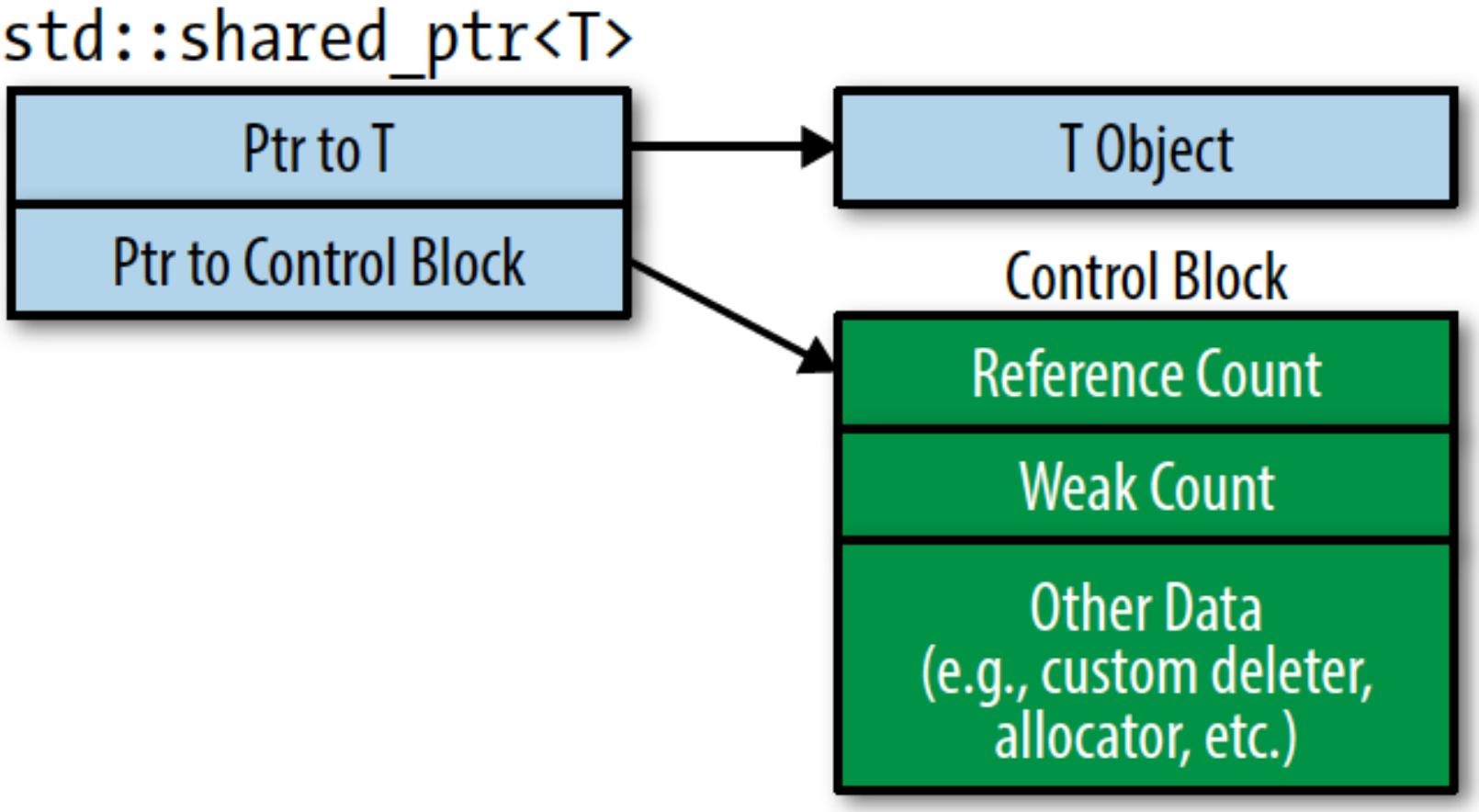
Custom Deleter (Cont…)

```
int main() {
    {
    auto pObj = get_object();
    std::cout << sizeof(pObj) << std::endl;
    }

    {
    auto pObjWithFuncDeleter =
             std::unique_ptr<MyObject, DeleterFunctionObject>(
                                    new MyObject, DeleterFunctionObject());
    std::cout << sizeof(pObjWithFuncDeleter) << std::endl;
    }

    {
    auto pObjWithFuncObject = std::unique_ptr<MyObject, void (*)(MyObject*)>(
        new MyObject, deleter_function);
    std::cout << sizeof(pObjWithFuncObject) << std::endl;
    }
}
```

Custom Deleter (Cont…)



Smart Pointers - Summary

- Use `std::unique_ptr` for exclusive-ownership resource management.
  - A return type from a Factory is a good example.
- Use `std::shared_ptr` for shared-ownership resource management.

Are raw pointers dead?

- Note that it doesn't mean that we are not going to use "regular" pointers anymore.
- **Raw pointers** are bad when used for performing manual memory management, i.e. allocating and deallocating objects through new and delete.
- When used purely as a means to achieve reference semantics and pass around non-owning, observing pointers, there is nothing intrinsically dangerous in raw pointers, except maybe for the fact that one should take care not to dereference a dangling pointer.

strongly typed enum

Strongly-typed enum

- History of enums in C and C++:

#define

```
# define TOP_LEFT        1
# define TOP_RIGHT       2
# define BOT_LEFT        3
# define BOT_RIGHT       4
```

C

```
enum windowCorner  {TOP_LEFT = 1, TOP_RIGHT,
                    BOT_LEFT, BOT_RIGHT};
enum direction  {TOPLEFT = 1, TOPRIGHT, BOTLEFT, BOTRIGHT};
```

C++

```
enum windowCorner  {TOP_LEFT = 1, TOP_RIGHT,
                    BOT_LEFT, BOT_RIGHT};
enum direction  {TOPLEFT = 1, TOPRIGHT, BOTLEFT, BOTRIGHT};
```

C++11

?

Strongly-typed enums (Cont…)

- **C++98 problem**: enum is not strong typing.

```
enum Animals_e{ BEAR, CAT, CHICHEN};
enum Birds_e{EAGLE, DUCK, CHICHEN};
                    //error! chicken has already declared!

int main()
{
    bool b1 = BEAR && DUCK; //what?!
}
```

Strongly-typed enums (Cont…)

- Too easy to convert the `enum` into `int` type:

```
enum Animals_e{ BEAR, CAT, CHICHEN};
enum Birds_e{EAGLE, DUCK, /*CHICHEN*/};
enum Colors_e{RED='r', BLUE='b', GREEN='g'};

void func(int i){
}

int main()
{
    func(BEAR); //too easy to convert
}
```

Strongly-typed enums (Cont…)

```
//trying to forward declaration
enum SomeEnum_e;

void func2(SomeEnum_e* s){
}
```

ISO C++ forbids forward references to 'enum' types

Strongly-typed enums (Cont…)

- In C++98 the underlying type of `enum` is always `int`. What if we want `long`? `char`?

```
enum Colors_e{RED='r', BLUE='b', GREEN='g'};

int main()
{
   std::cout<<"size of Animals_e is "<<sizeof(Animals_e)<<std::endl;
           //output: 4
}
```

Strongly-typed enums (Cont…)

- **C++11 solution**: C++11 introduces a new category of enums, called **strongly-typed enums** and the `enum class` keywords.
  - They no longer export their enumerators in the surrounding scope,
  - are no longer implicitly converted to integral types
  - can have a user-specified underlying type
  - can be forward declared

Strongly-typed enums (Cont…)

```
enum class Animals_e{ BEAR, CAT, CHICHEN};
enum class Birds_e{EAGLE, DUCK, /*CHICHEN*/};

enum class Colors_e : char {RED='r', BLUE='b', GREEN='g'};

void func(int i){
}

int main()
{
    bool b1 = Animals_e::BEAR && Birds_e::DUCK;
            //error: no match for 'operator&&'
    func(Animals_e::BEAR);
            //error: cannot convert 'Animals_e' to 'int'
    std::cout<<"size of Colors_e is "<<sizeof(Colors_e)<<std::endl;  //output: 1
}
```

Strongly-typed enums (Cont…)

- The new enums are "enum class" because they combine aspects of traditional enumerations (names values) with aspects of classes (scoped members and absence of conversions).

- Forward declaration of enums is possible since C++11.
  - Previously, the reason enum types couldn't be forward declared was because the size of the enumeration depended on its contents. C++ Compilers are allowed to specify the actual storage space based on the size necessary to contain all the values specified.
  - As long as the size of the enumeration is specified by the application, it can be forward declared.

Strongly-typed enums (Cont…)

```
// enum Enum1;       // Illegal in C++03 and C++11; no size is explicitly
                     // specified.
enum Enum2 : unsigned int;// Legal in C++11.
enum class Enum3;         // Legal in C++11, because enum class declarations
                          // have a default type of "int".
enum class Enum4: unsigned int;  // Legal C++11.

//trying to forward declaration
// enum SomeEnum_e;

//trying to forward declaration
enum class SomeEnum_e;

void func(SomeEnum_e s){
}

int main(){}
```

User Defined Literals

User Defined Literals

- **C++98** provides several literals:

```
int main(){
    //in c++ we have 4 basic categories of literals:
    //integers, floating points, characters and strings
    //we can add a suffix to specify the type
    auto x = 4;
    auto y = 4UL;
    auto z = "hello";
    auto k = 'a';
}
```

User Defined Literals (Cont…)

- **C++98 problem**: If we can add suffixes to the literals, why can't we use them to specify user-defined units for example?
  The suffix modifiers for literals are fixed by the C++ specification, one cannot create new literal modifiers.

- **The purpose**: make our code more readable and less error-prone.

```
long double height = 3.4;   //meters? cm? inches?
```

User Defined Literals (Cont…)

- The following code is **not supported** in the spec but can be much more readable:

```
//we wish to specify more data to know if conversions
//are needed:
long double height = 3.4cm;
long double ratio = 3.4cm / 2.1mm;
            //--> conversion is needed: 3.4*10 / 2.1
```

compile error!

- In addition, we wish that the unit translations will be performed at compile time, and not in runtime.

User Defined Literals (Cont…)

- **C++11 solution**: added **user-defined literals (UDL)**, which Allows integer, floating-point, character, and string literals to produce objects of user-defined type by defining a user-defined suffix.

- Thus, the code becomes more readable and conversion computations are done at compile time.

- The function called by a user-defined literal is known as **literal operator.** Only suffix form is supported.

User Defined Literals (Cont…)

- The name of the literal operator is **`operator""`** followed by the suffix.
- The suffix must begin with underscore.
- The first letter of the suffix cannot be uppercase. It is reserved, unless there is no space between **`operator""`** and the suffix:

```
long double operator""_KG(long double x){
    //OK. operator""  _KG is an error.
    return x;
}
```

User Defined Literals (Cont…)

```
// km
double operator"" _km( long double x ) { return x*1000; }

// m
double operator"" _m( long double x ) { return x; }

// millimeter
double operator"" _mm( long double x ){ return x / 1000; }

int main() {
    double distance = 4.5_km;
    std::cout << "(1) "<< distance << std::endl;
    std::cout << "(2) "<< 3.9_mm << std::endl;
    std::cout << "(3) "<< std::setprecision(8) << distance + 31.9_mm << std::endl;
    std::cout << "(4) "<< 45.9_km / 2.0_m << std::endl;
    std::cout << "(5) "<< 32.3_mm * 2.0_m << std::endl;
    return 0;
}
```

constexpr UDLs

A ▾ Save/Load + Add new... ▾ Vim CppInsights Quick-bench C++

```
6
7   // km
8   constexpr double operator"" _km( long double x ) {
9       return x*1000;
10  }
11
12  // millimeter
13  double operator"" _mm( long double x ){
14      return x / 1000;
15  }
16
17  constexpr long double operator""_KG(long double x){ //OK. operator""
18      return x;
19  }
20
21  int main()
22  {
23      double distance = 4.5_km;
24      double x = 4.5_mm;
25  }
26      // std::cout << "(1) "<< distance << std::endl;
27      // std::cout << "(2) "<< 3.9_mm << std::endl;
28      // std::cout << "(3) "<< std::setprecision(8) << distance + 31.9_
29      // std::cout << "(4) "<< 45.9_km / 2.0_m << std::endl;
30      // std::cout << "(5) "<< 32.3_mm * 2.0_m << std::endl;
31  //  return 0;
32  // }
```

x86-64 gcc 10.2 -std=c++17

A ▾ Output... ▾ Filter... ▾ Libraries + Add new... ▾ Add t

```
1   operator"" _mm(long double):
2           push    rbp
3           mov     rbp, rsp
4           fld     TBYTE PTR [rbp+16]
5           fld     TBYTE PTR .LC0[rip]
6           fdivp   st(1), st
7           fstp    QWORD PTR [rbp-8]
8           mov     rax, QWORD PTR [rbp-8]
9           movq    xmm0, rax
10          pop     rbp
11          ret
12  main:
13          push    rbp
14          mov     rbp, rsp
15          sub     rsp, 16
16          movsd   xmm0, QWORD PTR .LC2[rip]
17          movsd   QWORD PTR [rbp-8], xmm0
18          fld     TBYTE PTR .LC3[rip]
19          lea     rsp, [rsp-16]
20          fstp    TBYTE PTR [rsp]
21          call    operator"" _mm(long double)
22          add     rsp, 16
23          movq    rax, xmm0
24          mov     QWORD PTR [rbp-16], rax
```

Output (0/0) x86-64 gcc 10.2 - 1556ms (93061B)

COMPILER EXPLORER https://godbolt.org/z/8cjY3x

- **Note**: Since the introduction of user-defined literals, the code that uses format macro constants for fixed-width integer types with no space after the preceding string literal became invalid:

```
std::printf("%"PRId64"\n",INT64_MIN);
```

has to be replaced by

```
std::printf("%" PRId64"\n",INT64_MIN);
```

User Defined Literals (Cont…)

- maximal munch:

```
int foo = 1;
int bar = 0xE+foo;   // error, invalid preprocessing number 0xE+foo
int baz = 0xE + foo; // OK

int quux = bar+++++baz; // error: bar++ ++ +baz, not bar++ + ++baz.
```

- Thus, user-defined integer and floating-point literals ending in p, P, (since C++17) e and E, when followed by the operators + or -, must be separated from the operator with whitespace or parentheses in the source

```
auto x = 1.0_E+2.0;   // error
auto y = 1.0_a+2.0;   // OK
auto z = 1.0_E +2.0;  // OK
```

std::chrono_literals

- Since C++14 (and some since C++20) a few new standard literals were added:

D

d (since C++20)

H

h (since C++14)

M

min (since C++14)
ms (since C++14)

N

ns (since C++14)

S

s (since C++14)

U

us (since C++14)

Y

y (since C++20)

https://en.cppreference.com/w/cpp/symbol_index/chrono_literals

std::chrono_literals (Cont…)

```
#include <iostream>

#include <chrono>

using namespace std::chrono_literals;


int test_s()
{
    auto halfmin = 30s;
    std::cout << "half a minute is " << halfmin.count() << " seconds\n"
              << "a minute and a second is " << (1min + 1s).count() << " seconds\n";
}
```

Numeric Literals Separators

- Numeric literals of more than a few digits are hard to read.

- **C++14** define Simple Quotation Mark ' as a digit separator, in numbers and user-defined literals. This can make it easier for human readers to parse large numbers.

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'0011ll;
```

Numeric Literals Separators (Cont…)

- The position of the single quotes is irrelevant.
- All the following are equivalent:

```
long long a1 = 123456789ll;
long long a2 = 123'456'789ll;
long long a3 = 12'34'56'78'9ll;
long long a4 = 12345'6789ll;
```

- It is also allowed in user-defined literals:

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Raw String Literals

- **In C++98**, to escape characters like “\n” we use an extra “\”.

```
int main()
{
    //const char* fname1 = "c:\users\files\file.txt";
                        //error! tries to process \u and \f

    const char* fname2 = "c:\native\actions\team.txt";
                        //not sure we got the string we wanted...

    std::cout<<fname2<<std::endl;
}
```

Raw String Literals (Cont…)

- **Since C++ 11**, we can use raw strings in which escape characters (like \n \t or \") are not processed.
- The syntax of raw string is that the literal starts with R"(and ends in)".

```
int main(){
    const char* fname2 = "c:\native\actions\team.txt";
                        //not sure we got the string we wanted...
    std::cout<<fname2<<std::endl;

    const char* fname3 = R"(c:\native\actions\team.txt)";
    std::cout<<fname3<<std::endl;
}
```

Raw String Literals (Cont…)

- Raw string literals are often used in regular expressions that use character classes, and in HTML strings and XML strings.

```
#include <regex>

using namespace std;

int main() {
     regex email_pattern(R"(^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+
                            \.[a-zA-Z0-9-.]+$)");
}
```

https://gist.github.com/winhvu/91ee8795bc68be973c81efffb63d7dc4

Raw String Literals (Cont…)

- **problem**: What if we want to declare a string literal that includes the sequence)" inside it?

```
//what if I want to print ")" on screen, including the ""?
const char* parens = R("This course is called Modern C++ ("C++
                        11/14/17")";
            //Compilation Error!
std::cout<<parens<<std::endl;
```

Raw String Literals (Cont…)

- **solution**: add **delimiters** before and after the string literal:

```
const char* parens = R"d_d(This course is called Modern C++ ("C++11/14/17)")d_d";

std::cout<<parens<<std::endl;
```

std::string_view

std::string_view

- **Until C++17**: Manipulating std::string causes many times a separate copy of the string:

```
std::size_t length(const std::string& s)
{
    return s.size();
}

int main()
{
    return length("hallo world, long string");
}
```

COMPILER EXPLORER

Add... More

CppCast, the first podcast for C++ devs, by C++ devs

Sponsors intel PC-lint Share Other

C++ source #1

A Save/Load + Add new... Vim CppInsights Quick-bench C++

```cpp
#include <string>

std::size_t length(const std::string& s)
{
    return s.size();
}

int main()
{
    return length("hallo world, long string");
}
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++

x86-64 gcc 10.2 -std=c++1z -O3

A Output... Filter... Libraries + Add new... Add tool...

```asm
length(std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const&):
        mov     rax, QWORD PTR [rdi+8]
        ret
main:
        push    rbp
        xor     edx, edx
        push    rbx
        sub     rsp, 56
        lea     rdi, [rsp+16]
        lea     rbx, [rsp+32]
        mov     QWORD PTR [rsp+24]
        lea     rsi, [rsp+8]
        mov     QWORD PTR [rsp+16], rbx
        call    std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> >::_M_create(unsigned long&,
        mov     rdx, QWORD PTR [rsp+8]
        movdqa  xmm0, XMMWORD PTR .LC0[rip]
        movabs  rcx, 7453010373645639783
        mov     QWORD PTR [rsp+16], rax
        mov     QWORD PTR [rsp+32], rdx
        movups  XMMWORD PTR [rax], xmm0
        mov     rdx, QWORD PTR [rsp+16]
        mov     QWORD PTR [rax+16], rcx
        mov     rax, QWORD PTR [rsp+8]
        mov     QWORD PTR [rsp+24], rax
        mov     BYTE PTR [rdx+rax], 0
        mov     rdi, QWORD PTR [rsp+16]
        mov     rbp, QWORD PTR [rsp+24]
        cmp     rdi, rbx
        je      .L3
        mov     rax, QWORD PTR [rsp+32]
        lea     rsi, [rax+1]
        call    operator delete(void*, unsigned long)
.L3:
        add     rsp, 56
        mov     eax, ebp
        pop     rbx
        pop     rbp
        ret
```

https://compiler-explorer.com/z/oePfhYKh9

std::string_view (Cont…)

- **C++17** introduced more efficient way to manipulate strings: `std::string_view`.

- Unlike `std::string`, which keeps its own copy of the string, `std::string_view` provides a view of a string that is defined elsewhere.

- The purpose of `std::string_view` is to avoid copying data which is already owned by someone else and of which only a non-mutating view is required.

std::string_view (Cont…)

COMPILER EXPLORER

Add... More CppCast, the first podcast for C++ devs, by C++ devs

Sponsors intel PC-lint Solid Sands

Share Other

C++ source #1

C++

```cpp
#include <string>

std::size_t length(const std::string_view& s)
{
    return s.size();
}

int main()
{
    return length("hallo world, long string");
}
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++

x86-64 gcc 10.2 -std=c++1z -O3

Output... Filter... Libraries Add new... Add tool...

```asm
length(std::basic_string_view<char, std::char_traits<char> > const&):
        mov     rax, QWORD PTR [rdi]
        ret
main:
        mov     eax, 24
        ret
```

std::string_view (Cont…)

```
void testString(){
  char text[]{ "hello" };
  std::string str{ text };
  std::string more{ str };

  str+=" world";
  std::cout << text << " // " << str << " // " << more << '\n';
}

void testStringView(){
  char text[]{ "hello" }; // view the text "hello", which is stored in the binary
  std::experimental::string_view str{ text }; // view of the same "hello"
  std::experimental::string_view more{ str }; // view of the same "hello"

  text[0] = 'H';

  std::cout << text << ' ' << str << ' ' << more << '\n';
}
```

std::string_view (Cont…)

- When you use a `std::string_view`, it's best to avoid modifications to the underlying string for the remainder of the `std::string_view`'s life to prevent confusion and errors.

- Use `std::string_view` instead of C-style strings.

- Prefer `std::string_view` over `std::string` for read-only strings, unless you already have a `std::string`.

The substr method

- `std::string` and `std::string_view` have both a method substr. The method of the `std::string` returns a substring but the method of the `std::string_view` returns a view of a substring.

- `std::string::substr` has linear complexity. `std::string_view::substr` has constant complexity.

std::string_view (Cont…)

- Unlike C-style strings and std::string, `std::string_view` doesn't use null terminators to mark the end of the string. Rather, it knows where the string ends because it keeps track of its length.

```
int main(){
  char vowels[]{ 'a', 'e', 'i', 'o', 'u' };   // No null-terminator

  // vowels isn't null-terminated. We need to pass the length manually.
  // Because vowels is an array, we can use std::size to get its length.
  std::string_view str{ vowels, std::size(vowels) };

  std::cout << str << '\n'; // This is safe. std::cout knows how to print std::string_views.

  return 0;
}
```

std::string_view (Cont…)

- Being only a view, a `std::string_view`'s lifetime is independent of that of the string it is viewing.
  - If the viewed string goes out of scope, `std::string_view` has nothing to observe and accessing it causes undefined behavior.

- The same can happen when we create a `std::string_view` from a std::string and modify the `std::string`.
  - Modifying a `std::string` can cause its internal string to die and be replaced with a new one in a different place. The `std::string_view` will still look at where the old string was, but it's not there anymore.

Converting a std::string_view to a std::string

- An `std::string_view` will not implicitly convert to a std::string, but can be explicitly converted:

```
void func(std::string s)
{}

int main(){
  std::experimental::string_view sv{ "hello" };

//   func(sv); // compile error: won't implicitly convert

  std::string str{ sv }; // OK
  func(str); // OK

  func(static_cast<std::string>(sv)); // OK
}
```

Converting a std::string_view to a C-style string

- Some old functions still expect C-style strings.
- To convert a `std::string_view` to a C-style string, we can do so by first converting to a `std::string`:

```
void convertToCStyleString(){
  std::experimental::string_view sv{ "hello" };

  std::string str{ sv };

  const char* szNullTerminated{ str.c_str() };

  // Pass the null-terminated string to the function that we want to use.
  std::cout << str << " has " << std::strlen(szNullTerminated) << " letter(s)\n";
}
```

unordered STL containers

Unordered STL Containers

- The set and map containers has since **C++11** an unordered version, which is a hash-based implementation (instead of the tree-based version).

- In theory, hash-tables offer constant-time insert, add and delete operations. However, in practice the difference between the O(log n) operations of the tree-based containers and the unordered version might not be so obvious, especially if you are not storing a very large number of elements in your container.

std::unordered_map

- Unordered maps are associative containers that store elements formed by the combination of a key value and a mapped value, and which allows for fast retrieval of individual elements based on their keys.

- Both key and value can be of any type predefined or user-defined.

std::unordered_map (Cont…)

```
#include <iostream>
#include <unordered_map>

int main(){
    std::unordered_map<std::string, int> umap;
    // inserting values by using [] operator
    umap["A"] = 10;
    umap["B"] = 20;
    umap["C"] = 30;

    // Traversing an unordered map
    for (auto x : umap)
        std::cout << x.first << " " <<
                x.second << std::endl;
}
```

```
int main(){
    std::unordered_map<std::string, double> umap = {
        {"One", 1}, {"Two", 2}, {"Three", 3}
    };
    umap["PI"] = 3.14;
    umap.insert({"e", 2.718});

    std::string key = "PI";
    std::cout<<key<<((umap.find(key) == umap.end()) ?
                                    " not found\n" : "Found ")<<std::endl;

    key = "lambda";
    std::cout<<key<<((umap.find(key) == umap.end()) ?
                                    " not found\n" : "Found ")<<std::endl;

    // iterating over all value of umap
    std::cout << "\nPrint All Elements : \n";
    for (const auto& [the_key,the_value]: umap){
        std::cout << the_key << " " << the_value << std::endl;
    }
}
```

Hands-On Exercise

- We will implement a LRU caching system.
  - LRU = a policy to remove the least recently used frame when the cache is full.

- **We use two data structures to implement an LRU Cache:**
  - **Queue** which is implemented using a doubly linked list. The maximum size of the queue will be equal to the total number of frames available (cache size). The most recently used pages will be near front end and least recently pages will be near the rear end.
  - **A Hash** with page number as key and address of the corresponding queue node as value.

std::unordered_set

```
#include <iostream>
#include <unordered_set>

int main() {

  // uniform initialization
  std::unordered_set<int> numbers {1, 100, 10, 70, 100};

  std::cout << "numbers = ";
  for(auto &num: numbers) {
    std::cout << num << ", ";
  }

  return 0;
}
```

Hands-On Exercise

- Given an array(list) of integers, find all the duplicates among them.

User Define Hash Functions

- The STL provides hash functions for commonly used types, like string and numeric values such as int, meaning that we won't have to provide any hash functions explicitly when creating an unordered container instance.

- However, when we want to hold a more complex type, or even a user-defined class, C++ unordered containers would fail to find a hash function for itself to use.

User Define Hash Functions (Cont…)

```
struct PairHash {
  template <typename T1, typename T2>
  auto operator()(const std::pair<T1, T2> &p) const -> size_t {
    return std::hash<T1>{}(p.first) ^ std::hash<T2>{}(p.second);
  }
};

void unordered_with_functors() {
    std::unordered_set<std::pair<std::string, int>, PairHash> uset;
    std::unordered_map<std::pair<double, std::string>, int , PairHash> umap;
}
```

Parallel STL

Parallel STL

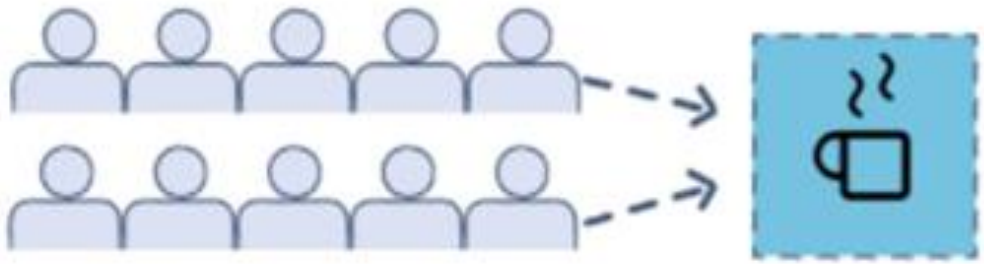
- **C++17** added overloads to the <algorithms> library that can now executed parallelly.
  - The elements will be split to different threads.

Parallel STL (Cont…)

- Concurrency and parallelism often get mixed up.

- In **parallelism**, we run multiple copies of the same program simultaneously, but they are executed on different data.
- **Concurrent** programming involves a shared memory location, and the different threads actually “read” the information provided by the previous threads.

Concurrent: *Two Queues* & a *Single* Espresso machine.



Parallel: *Two Queues* & *Two* Espresso machines.

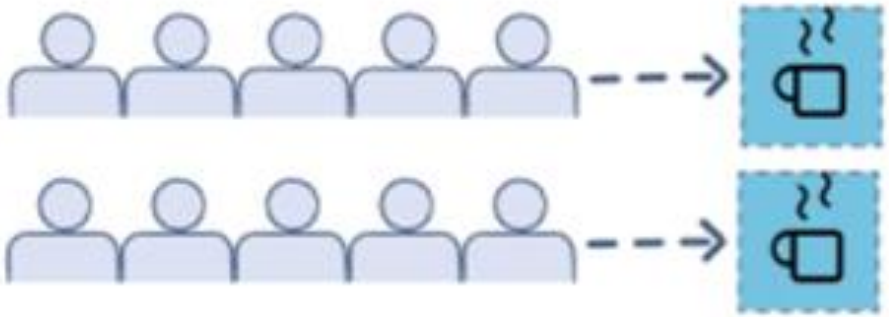


Image from EdPresso shot, "What is concurrent programming?"

Parallel STL (Cont…)

- **Since C++17**, STL was added parallel versions of STL algorithms.
- They are overloading of the current algorithms with a first parameter signifying the execution policy, which exists in the `<execution>` header.
  - Each one is a different class for the overloading to work.

Parallel STL (Cont…)

- `std::execution::sequenced_policy` – the algorithm's operations will be performed sequentially.
  Send the object `std::execution::seq` as parameter to use it.

- `std::execution::parallel_policy` – the algorithm should be executed parallelly.
  Send the object `std::execution::par` as parameter to use it.

- `std::execution::parallel_unsequenced_policy` – the execution may be parallelized, vectorized or migrated across threads.
  Send the object `std::execution::par_unseq` as parameter to use it.

Parallel STL (Cont…)

```
// standard sequential sort
sort(v.begin(), v.end());

// explicitly sequential sort
sort(execution::seq,v.begin(), v.end());

// permitting parallel execution
sort(execution::par,v.begin(), v.end());

// permitting parallel execution and vectorization as well
sort(execution::par_unseq,v.begin(), v.end());
```

MKBN

```
int main() {
    std::random_device rd;

    // generate some random doubles:
    const size_t testSize = 1'000'000;
    std::vector<double> doubles(testSize);
    for (auto& d : doubles) {
        d = static_cast<double>(rd());
    }
    //-----------------------------------------
    std::vector<double> sorted(doubles);

    std::sort(sorted.begin(), sorted.end());

    std::sort(std::execution::par_unseq,
                sorted.begin(), sorted.end());
}
```

Parallel STL (Cont…)

- We should remember that the parallel algorithms have an overhead of starting the thread and managing the merge at the end.

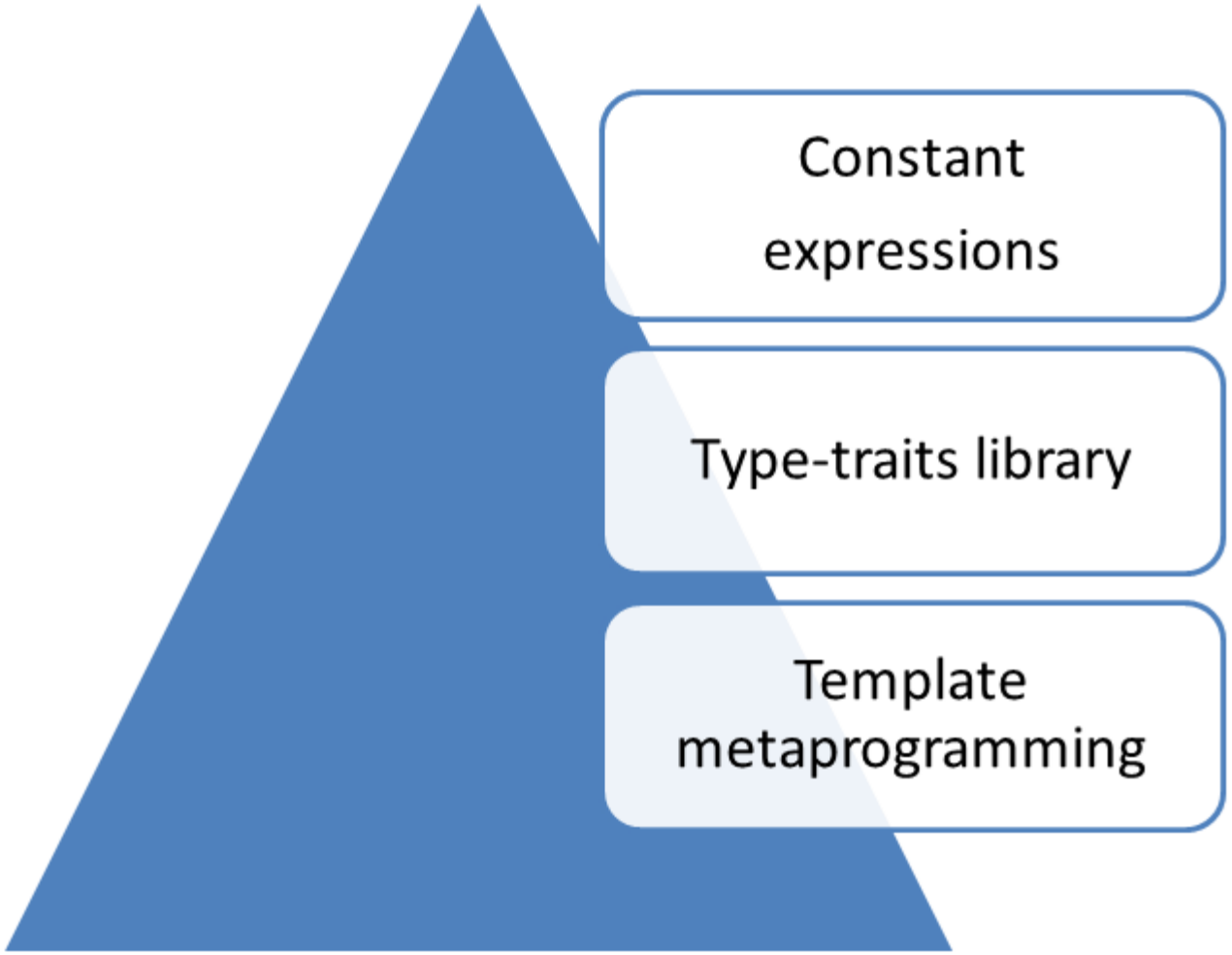
- **Note**: When using parallel execution policy, it is the programmer's responsibility to avoid data races and deadlocks.

```
int a[] = {0,1};
std::vector<int> v;
std::for_each(std::execution::par,
                std::begin(a), std::end(a), [&](int i) {
                     v.push_back(i*2+1); // Error: data race
                });
```

```
std::atomic<int> x{0};
int a[] = {1,2};
std::for_each(std::execution::par,
                std::begin(a), std::end(a), [&](int) {
                  x.fetch_add(1, std::memory_order_relaxed);
                  while (x.load(std::memory_order_relaxed) == 1)
                  { } // Error: assumes execution order
                });
//-----------------------------------------------------------
//using mutex:
int x = 0;
std::mutex m;
int a[] = {1,2};
std::for_each(std::execution::par,
           std::begin(a), std::end(a), [&](int) {
             std::lock_guard<std::mutex> guard(m);
             ++x; // correct
           });
```

Compile Time Programming

Programming at Compile Time



macro metaprogramming

```
#define CREATE_OPERATOR_DEC(OP)\
  FP_Number& operator OP##=(const FP_Number& x);

#define CREATE_OPERATOR_IMP(OP) \
  FP_Number& FP_Number::operator OP##= (const FP_Number& x) \
  {     _d = _d OP x._d; return *this;   }

class FP_Number{
public:
  FP_Number(double d) : _d(d){}

  FP_Number operator+(const FP_Number& fpNum) const{
    return FP_Number(fpNum._d + _d);
  }
  //... rest of operators...

  CREATE_OPERATOR_DEC(+)
  CREATE_OPERATOR_DEC(-)
  CREATE_OPERATOR_DEC(*)
  CREATE_OPERATOR_DEC(/)
public:
  double get() const {return _d;}
private:
  double _d;
};

CREATE_OPERATOR_IMP(+)
CREATE_OPERATOR_IMP(-)
CREATE_OPERATOR_IMP(*)
CREATE_OPERATOR_IMP(/)
```

Template Metaprogramming

- **Template Metaprogramming** is a technique in which templates are used by the compiler to generate temporary source code.

- At runtime we use data and functions.
  At compile time we use meta-data and meta-functions.
  - **Metadata**: Types and integral values that are used in meta-functions.
  - **Meta-function**: Functions that are executed a compile time.

Template Metaprogramming (Cont…)

- **Building Blocks**:
  - **variables** = The 'variables' in TMP are not really variables since their values cannot be altered.
  - **functions** = A function maps one or more input parameters into an output value. The TMP analogue to this is a template class.
  - **branching** = A conditional branch can be constructed by writing two alternative specializations of a template class.
  - **recursion (loop)** = a template class whose output value recursively refers to itself, and one or more specializations that give fixed values to prevent infinite recursion.

Template Metaprogramming (Cont…)

- **variables** in TMP:

```
struct ValueHolder
{
    enum { value = 2 };
};
```

Template Metaprogramming (Cont…)

- **functions** in TMP:

```
template<int X, int Y>
struct Adder
{
    enum { result = X + Y };
};
```

Template Metaprogramming (Cont…)

- **branching** in TMP:

```
template <bool isEven>
struct if_even_;

template <>
struct if_even_<true>{
    enum{result=true};
};

template <>
struct if_even_<false>{
    enum{result=false};
};

int main(){
    std::cout<<if_even_<(12%2==0)>::result;
}
```

https://godbolt.org/z/aq4T7P

Template Metaprogramming (Cont…)

- **recursion** (loop):

```
template <unsigned n>
struct factorial
{
  enum { value = n * factorial<n-1>::value };
};

template <>
struct factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int array[ factorial<7>::value ];
}
```

- Notice that in TMP, Instead of modifying a value you return each time a new value.

Metaprogramming: Hands-on exercise

- Write a meta-function that gets 2 unsigned integers a,b and prints the number of numbers in the range [1,…,a] that are divided by b.

```
int main()
{
    std::cout<<"#numbers till 20 divided by 3 is: "<<
        count_div_by_b_numbers_till_a<20,3>::value <<
        std::endl;
}
```

- Check your code in compiler explorer to make sure all is calculated at compile time.

Template Metaprogramming (Cont…)

- It is possible to use “variables” also to define a type:

```
struct TypeHolder
{
    typedef int type;
};

void testVar()
{
    TypeHolder::type v = 3;
}
```

Template Metaprogramming (Cont…)

```
template<typename T>
struct removeConst{
    typedef T type;
};

template<typename T>
struct removeConst<const T> {
    typedef T type;
};


int main(){

    std::is_same<int, removeConst<int>::type>::value;        // true
    std::is_same<int, removeConst<const int>::type>::value;  // true

}
```

Metaprogramming: Hands-on exercise

- Implement the meta-function that checks if the T type is a pointer or not:

```
int* testFunc(){
    static int x;
    return &x;
}

int main(){
    int x;
    int* p = &x;
    const int* pc = &x;

    std::cout<<"is x a pointer? "<<is_pointer_check(x)<<std::endl;
    std::cout<<"is x a pointer? "<<is_pointer_check(p)<<std::endl;
    std::cout<<"is x a pointer? "<<is_pointer_check(pc)<<std::endl;
    std::cout<<"is x a pointer? "<<is_pointer_check(testFunc())<<std::endl;
}
```

```cpp
template< class T >
struct is_pointer{
  static const bool value = false;
};

template< class T >
struct is_pointer< T* >{
  static const bool value = true;
};

template<class T>
bool is_pointer_check(T x){
    return is_pointer<T>::value;
}
```

```cpp
int* testFunc(){
    static int x;
    return &x;
}

int main()
{
    int x;
    int* p = &x;
    const int* pc = &x;

    std::cout<<is_pointer_check(x)<<std::endl;
    std::cout<<is_pointer_check(p)<<std::endl;
    std::cout<<is_pointer_check(pc)<<std::endl;
    std::cout<<is_pointer_check(testFunc())<<std::endl;
}
```

Type Traits

- **C++98 problem**: if the implementation is depended on T, it is very difficult to **grab any information on T, even if it is known to the compiler**. There are just too many options to check.
  - Is it an integer? Is it a function? Is it a pointer? Or maybe a class? Does it have a destructor? Can you copy it? Will it throw exceptions? ... and so on.

Type Traits (Cont…)

- **C++11 solution**: type traits library `<type_traits>` was added.
- The type traits supports type checks, type comparisons, and type modifications at compile time.

```
#include <iostream>
#include <type_traits>

class MyClass {};

int main(){
    std::cout << std::is_floating_point<MyClass>::value << '\n';
    std::cout << std::is_floating_point<float>::value << '\n';
    std:cout << std::is_floating_point<int>::value << '\n';
}
```

Type Traits (Cont…)

https://en.cppreference.com/w/cpp/header/type_traits

Primary type categories

`is_void` (C++11)	checks if a type is `void` (class template)
`is_null_pointer` (C++14)	checks if a type is `std::nullptr_t` (class template)
`is_integral` (C++11)	checks if a type is an integral type (class template)
`is_floating_point` (C++11)	checks if a type is a floating-point type (class template)
`is_array` (C++11)	checks if a type is an array type (class template)
`is_enum` (C++11)	checks if a type is an enumeration type (class template)
`is_union` (C++11)	checks if a type is an union type (class template)
`is_class` (C++11)	checks if a type is a non-union class type (class template)
`is_function` (C++11)	checks if a type is a function type (class template)
`is_pointer` (C++11)	checks if a type is a pointer type (class template)
`is_lvalue_reference` (C++11)	checks if a type is a *lvalue reference* (class template)
`is_rvalue_reference` (C++11)	checks if a type is a *rvalue reference* (class template)
`is_member_object_pointer` (C++11)	checks if a type is a pointer to a non-static member object (class template)
`is_member_function_pointer` (C++11)	checks if a type is a pointer to a non-static member function (class template)

Type Traits (Cont…)

- type traits is a way to restrict templates to types that have certain properties.
- By convention, traits are always implemented as structs.
- Another convention is that the structs used to implement traits are known as **trait classes**.

Type traits examples

```
std::is_same<int, std::remove_const<int>::type>::value;        // true
std::is_same<int, std::remove_const<const int>::type>::value;  // true
```

```
struct A { };
struct B { B& operator= (const B&) = delete; };

int main() {
  std::cout << std::boolalpha;
  std::cout << "is_copy_assignable:" << std::endl;
  std::cout << "int: " << std::is_copy_assignable<int>::value << std::endl;
  std::cout << "A: " << std::is_copy_assignable<A>::value << std::endl;
  std::cout << "B: " << std::is_copy_assignable<B>::value << std::endl;
  return 0;
}
```

Type Traits (Cont…)

	primary categories	composite categories		
fundamental	void			
	`std::nullptr_t`		scalar	object
	integral	arithmetic		
	floating point			
compound	pointer			
	member object pointer	member pointer		
	member function pointer			
	enum			
	union			
	class*			
	array			
	l-value reference	reference		
	r-value reference			
	function			

* = excluding unions

https://www.cplusplus.com/reference/type_traits/

type_traits Hands-on Exercise

- Look at the <type_traits> header file and find how to implement the following requirements:
  - create a template function where the type T must derive from a base.
  - create a static assert which makes sure the type is both defaulted and copy constructable.

type_traits Hands-on Exercise

- Implement the following function:

```
template<class T>
T* GetRawPtr(T&& t){
    //if it is a pointer - just return it

    //if it is a smart pointer - return its underlined pointer

    //otherwise - return null
}
```

- You will need to google a bit… You can also use this hint.

static_assert

- **problem**: With type_traits we can check if an argument is fulfilling some conditions, but the check is done on runtime.

```
template <typename T>
T div(T a, T b){
    //we wish to divide a and b only if they are floating point types
    if(std::is_floating_point<T>::value==false){
        std::cout<<"div::error::can be used only with floating point types\n";
        return 0;
    }
    return a/b;
}
```

static_assert(Cont…)

- In **C++11** we have a new king of assert to evaluate expressions: `static_assert`.
- Used to test expressions at compile time.
- If the expression's value is false, the compiler generates an error with the string provided as an argument to the `static_assert`.
- It is widely used with type traits.

static_assert(Cont…)

```
template <typename T>
T mydiv(T a, T b){
    //we wish to divide a and b only if they are floating point types
    static_assert(std::is_floating_point<T>::value,
                "div::error::can be used only with floating point types\n");
     return a/b;
}

int main(){
    mydiv(2,4);
}
```

```
type named 'type' in 'struct std::enable_if<false, std::basic_ostream<char>&>'
.\static_assert.cpp: In instantiation of 'T mydiv(T, T) [with T = int]':
.\static_assert.cpp:30:14:   required from here
.\static_assert.cpp:18:19: error: static assertion failed: div::error::can be used only with floating point types

     static_assert(std::is_floating_point<T>::value,
```

static_assert(Cont…)

- **Since C++17** the `static_assert` does not require the message as the second parameter.
  It became optional.

- The error message will be general as `static_assert` failed.

SFINAE

- "**Substitution Failure Is Not An Error**“
  - Substitution = replacing the template arguments with the real type.
  - Sometimes when creating candidates from template functions, some options are not valid. This rule says to take off those candidates from the candidates set instead of generating an error.

```
template <typename T>
typename T::value_type negate(const T& t) {
  return -t();
}

//if T = int then we get:
// int::value_type ?!
```

SFINAE (Cont…)

- The C++11 standard states that when a substitution failure occurs, type deduction for this particular type fails. The compiler simply ignores this candidate and looks at the others.
  That's SFINAE.

- The idea was so common that it was inserted to the C++11 standard as `std::enable_if` statement.
  - It was part of the boost library for many years and imported from there.

std::enable_if

- `std::enable_if`:

```
template< bool B, class T = void >
struct enable_if;
```

  If `B` is true, `std::enable_if` has a public member typedef type equal to T (std::enable_if::type = T); otherwise, there is no member typedef (std::enable_if::type is not defined).

- `std::enable_if` can be used as an additional function argument (not applicable to operator overloads), as a return type (not applicable to constructors and destructors), or as a class template or function template parameter.

std::enable_if (Cont…)

- The **enable_if** family of templates is a set of tools to allow a function template or a class template specialization to include or exclude itself from a set of matching functions or specializations based on properties of its template arguments.

```
template<bool Cond, class T = void> struct enable_if {};
template<class T> struct enable_if<true, T> { typedef T type; };
```

std::enable_if (Cont…)

- The old way:

```
template<class T>
bool is_odd(T i){
    //T can be anything. even structs that has
    //nothing to do with odd...
    //how can we prevent this?
}
```

std::enable_if (Cont…)

```
// 1. the return type (bool) is only valid if T is an integral type:
template <class T>
typename std::enable_if<std::is_integral<T>::value,bool>::type
is_odd (T i) {
      return bool(i%2);
}

// 2. the second template argument is only valid if T is an integral type:
template < class T,
             class=typename std::enable_if<std::is_integral<T>::value>::type>
bool is_even (T i) {
      return !bool(i%2);
}
```

std::conditional

- **Added in C++11**.
  Defined in header <type_traits>:

```
template< bool B, class T, class F >
struct conditional;
```

- Provides member typedef type, which is defined as T if B is true at compile time, or as F if B is false.

std::conditional (Cont…)

```
int main()
{
    typedef std::conditional<true, int, double>::type Type1;
    typedef std::conditional<false, int, double>::type Type2;
    typedef std::conditional<sizeof(int) >= sizeof(double), int, double>::type Type3;

    std::cout << typeid(Type1).name() << '\n';
    std::cout << typeid(Type2).name() << '\n';
    std::cout << typeid(Type3).name() << '\n’;
}
```

std::conditional (Cont…)

- Example: check if the system is little or big endian.

- In order to know how many bits to swap, we can use `std::is_conditional` that depends on the T's size to call the right function.

Template Metaprogramming (Cont…)

- To conclude, we saw metaprogramming in C++ using templates.

- There is another way to write metaprogramming in C++, and it is by using `constexpr`.
  - considered more readable and intuitive and even more efficient.

constexpr

constexpr

- **Constant expressions** are optimization opportunities for compilers, and compilers frequently execute them at compile time and hardcode the results in the program.

- Defining an array requires a constant expression as its size, and enumerator values must be constant expressions.

```
const int val = 3;
enum {FIRST = 0, SECOND = 1, THIRD=val};

int arr[val];
```

constexpr (Cont…)

- **C++98 problem**: we cannot use functions that always returns a constant value as a constant expression.

```
int function() { return 5; }

//int arr2[function()];
     //won't compile, even if function always returning 5
```

constexpr (Cont…)

- **C++11 solution**: use the `constexpr` keyword and tell the compiler to compile it at compilation phase.

```
constexpr int function() { return 5; }

int arr2[function()];
```

constexpr (Cont…)

- `constexpr` indicates that the value, or return value, is constant and, where possible, is computed at compile time.

SYNTAX

```
constexpr literal-type identifier = constant-expression ;
constexpr literal-type identifier { constant-expression } ;
constexpr literal-type identifier ( params ) ;
constexpr ctor ( params ) ;
```

- The parameters can be one or more, each of which must be a literal type and must itself be a constant expression.

- A `constexpr` variable or function **must return a literal type**.

constexpr (Cont…)

- `constexpr` represents an expression that is constant, that will possibly be **evaluated at compile time**.
- Can be applied to **variables** or **functions**.
- Different from `const` type since **const type not always evaluated at compile time**.

```
int getNumber_nc() { return 5; }

const int i = getNumber_nc();
     //i will be initialized at run time and not compile time!
```

constexpr (Cont…)

- `constexpr` is more strict that `const`:

```
int a;
const int b = a;//correct
constexpr int c = a;//wrong, because a has no value yet
```

constexpr (Cont…)

- A `constexpr` specifier used in an object declaration or non-static member function **implies const**.
- Such an object must be of a literal type and **initialized with a constant expression**.

```
int main()
{
    //constexpr int v; //error! must be initialized.
    constexpr int v = 3; //OK
}
```

constexpr (Cont…)

- `constexpr` can be initialized only with constant values that are evaluated at compile time.

```
int getNumber_nc() { return 5; }
constexpr int getNumber() { return 5; }

constexpr int j = getNumber_nc(); //ERROR!
constexpr int r = getNumber(); //OK
```

constexpr (Cont…)

- A `constexpr` specifier used in a function or static member variable (since C++17) declaration implies **inline**.
- If any declaration of a function or function template has a `constexpr` specifier, then every declaration must contain that specifier.
- `constexpr` function must not be virtual (until C++20).
- each of its parameters and its return value must be a literal type.

constexpr (Cont…)

- If we initialize a **non-const variable** with a `constexpr` function, then the function is **evaluated at runtime**:

```
constexpr int r = getNumber();

int q = getNumber();
     //getNumber will be called at runtime and not compile time
```

constexpr (Cont…)

- **C++11 problem**: `constexpr` functions can only contain **a single expression that is returned**.

- In the following example, if the compiler can calculate the return value at compile-time, why we should pay for it at runtime?

```
constexpr int function3(){
    int x = 3;
    return 3;
}
```

This code will not compile in C++11!!!

constexpr (Cont…)

```
size_t factorial(int n){
    if(n==1){
        return 1;
    }
    return factorial(n-1)*n;
}

int main()
{
    size_t result = factorial(20);
    std::cout<<"factorial(20) = "<<result<<std::endl;
}
```

will be computed at runtime!

- **C++14 solution**: `constexpr` function can have more than one return statements.

```
constexpr size_t factorial(int n){
    if(n==1){
        return 1;
    }
    return factorial(n-1)*n;
}

int main(){
    size_t result = factorial(20);
    std::cout<<"factorial(20) = "<<result<<std::endl;
}
```

will be computed at compilation time!

constexpr (Cont…)

- `constexpr` functions in **C++14**:
  - can invoke other `constexpr` functions.
  - can have variables which have to be initialized by a constant expression.
  - can have conditional expressions or loops.
  - are implicit inline.
  - cannot have static or thread_local data.

constexpr (Cont…)

- In a `constexpr` function**, the parameter passed should be `constexpr` or `const`**. Otherwise, the function will be calculated at runtime.

```
constexpr size_t factorial(int n){
    if(n==1){
        return 1;
    }
    return factorial(n-1)*n;
}

int main()
{
    size_t result = factorial(20);

    int x = 4;
    result = factorial(x);
}
```

Try to run it with Compiler Explorer:

https://godbolt.org/z/Trb57P

COMPILER EXPLORER

Add... More

CppCast, the first podcast for C++ devs, by C++ devs

Sponsors intel PC-lint

Share Other

C++ source #1

C++

```cpp
#include <iostream>

constexpr size_t factorial(int n){
    if(n==1){
        return 1;
    }
    return factorial(n-1)*n;
}

int main()
{
    size_t result = factorial(20);

    int x = 5;
    result = factorial(x);
}
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++

x86-64 gcc 10.2

Compiler options...

```asm
        imul    rax, rdx
.L3:
        leave
        ret
main:
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        movabs  rax, 2432902008176640000
        mov     QWORD PTR [rbp-8], rax
        mov     DWORD PTR [rbp-12], 5
        mov     eax, DWORD PTR [rbp-12]
        mov     edi, eax
        call    factorial(int)
        mov     QWORD PTR [rbp-8], rax
        mov     eax, 0
        leave
        ret
__static_initialization_and_destruction_0(int, i
        push    rbp
```

Output (0/0) x86-64 gcc 10.2 - 1249ms (923988)

Read the new cookie policy Compiler Explorer uses cookies and other related techs to serve you Consent Don't

constexpr constructor

- Previously, only expressions of built-in types could be valid constant expressions.
- With `constexpr` constructors, objects of user-defined types can be included in valid constant expressions.

- A `constexpr` constructor allows the compiler to **initialize the object at compile-time**, provided that the constructor's arguments are all constant expressions.

constexpr constructor (Cont…)

- Formally, a constant expression constructor is one that meets the following criteria:
  - It's declared `constexpr` explicitly.
  - It can have a member initialization list involving only potentially constant expressions (if the expressions used aren't constant expressions, then the initialization of that object will be dynamic).
  - Its body must be empty.

```
struct complex{
    //a constant expression constructor
    constexpr complex(double r, double i) : re(r), im(i) { }//empty body
    //constant expression functions
    constexpr double real() const { return re; }
    constexpr double imag() const { return im; }
private:
    double re;
    double im;
};

int main(){
    constexpr complex COMP(0.0, 1.0); // creates a literal complex

    double x = 1.0;
    // constexpr complex cx1(x, 0); // error: x isn't a constant expression

    const complex cx2(x, 1); //OK, dynamic initialization

    constexpr double xx = COMP.real(); // OK
    constexpr double imaglval = COMP.imag(); //OK, static init

    complex cx3(2, 4.6); //dynamic initialization
}
```

C++17: constexpr if – Compile-Time If

- **C++17** added a new feature that enables it to conditionally compile source code.
- It discards branches of an if-statement at compile-time.

```
template <typename T>
auto get_value(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t;  // deduces return type to int for T = int
}
```

C++17: constexpr if – Compile-Time If (Cont…)

- The discarded statements must be still valid statements.
- if `constexpr` can be used only inside functions.

```
template <typename T>
void printData(T& t){
    if constexpr (std::is_array_v<T>){
        for(auto x : t){
            std::cout<< x <<" ";
        }
        std::cout<<std::endl;
    }
    else if constexpr (std::is_pointer_v<T>){
        std::cout<<*t<<std::endl;
    }
    else{
        std::cout<<t<<std::endl;
    }
}

int main(){
    int arr[]{1,2,3,4,5};
    printData(arr);
}
```

Template Metaprogramming versus constexpr Functions

Characteristics	Template Metaprogramming	`constexpr` Functions
Execution time	Compile time	Compile time and runtime
Arguments	Types and values	Values
Programming paradigm	Functional	Imperativ
Modification	No	Yes
Control structure	Recursion	Conditions and loops
Conditional execution	Template specialisation	Conditional statements

https://www.modernescpp.com/index.php/c-core-guidelines-programming-at-compile-time-with-constexpr

Advantages of constexpr functions

- `constexpr` functions are more comfortable to write and to maintain than meta-functions.
- can run at compile time and runtime.
- can deal with floating point numbers. TMP accepts only integral numbers.

Lambda Expressions

Lambdas

- Lambdas are “anonymous functions”.
- It is a powerful feature borrowed from functional programming.
- A lambda function is a function that you can write inline in your source code (usually to pass in to another function, similar to the idea of a functor or function pointer).

Lambda - motivation

```
int main(){
    std::vector<int> vec{1,2,3,4,5,6,7,8};
    size_t s = std::count_if(vec.begin(), vec.end(), ??? );
    std::cout << "# of odd integers: " << s << "\n";
}
```

Lambda – motivation (cont…)

- **Option 1: using a function ptr or a function object.**

- **Function objects**:
  - class with constructor and single member operator()

```
template <class T>
class myFunc{
public:
   myFunc( /*arguments save needed state info */ ) { }
   T operator()(/* args for func obj */) {
      /*
      call some useful function with saved
      state info and args as its parameters
      */
   }
private:
   /* state info here */
```

Lambda – motivation (cont…)

```
bool isOdd(int x){
    return x%2==1;
}

struct isOddF{
    bool operator()(int x){ return x%2==1; }
};

int main(){
    std::vector<int> vec{1,2,3,4,5,6,7,8};

    //option 1: pointer to function
    size_t s = std::count_if(vec.begin(), vec.end(),  isOdd);
    std::cout << "# of odd integers: " << s << "\n";

    //option 2: function object
    s = std::count_if(vec.begin(), vec.end(),  isOddF());
    std::cout << "# of odd integers: " << s << "\n";
}
```

Lambda – motivation (cont…)

```
struct isOddF{
    bool operator()(int x){ return x%2==1; }
};

struct isEvenF{
    bool operator()(int x){ return x%2==0; }
};

struct isPosF{
    bool operator()(int x){ return x>0; }
};

struct isNegF{
    bool operator()(int x){ return x<0; }
};

int main(){
    std::vector<int> vec{1,2,3,4,5,6,7,8};

    s = std::count_if(vec.begin(), vec.end(),  isOddF());
    std::cout << "# of odd integers: " << s << "\n";
}
```

Lambda – motivation (cont…)

- **Option 2: using lambda expressions.**

Lambdas (Cont...)

- With Lambdas it is much easier:

capture specification

```
int main()
{
    auto func = []() { std::cout << "Hello World!\n"; };
    func(); //call this lambda
}
```

Lambdas (Cont…)

- Syntax:

```
[](<args>)<mutable><excp specification> -> <return type>
{

}
```

Lambdas (Cont...)

```
struct isOddF{
    bool operator()(int x){
        return x%2==1;
    }
};

int main()
{
    std::vector<int> vec{ 1,2,3,4,5,6,7,8 };
    size_t s = std::count_if(vec.begin(), vec.end(), [](int x){ return x%2; });
    std::cout << "# of odd integers: " << s << "\n";

}
```

Return value from Lambda Expression

- **In C++11**, if the compiler can deduce the return value of the lambda function, it will do it rather than force you to include it.

```
int main()
{
    auto max = [](int x, double y) -> int{
        if(x>y){
            return x;
        }
        return y;
    };
}
```

Lambda's capture section

- If we want to send local variables as parameters to the lambda expression, we use the capture section:

- To capture nothing – use [ ].
- To capture all of the scope variables by value, use [=].
- To capture all of the scope variables by reference, use [&].

Example: counting numbers who's divided by x

```
//C++98 syle:
//we need to declare a functor and send it to std::count_if
struct isMultOf{
    isMultOf(int m) : _mult{m}{}
    bool operator()(int x){
        return !(x%_mult);
    }
private:
    int _mult;
};

int main(){
    std::vector<int> vec{ 1,2,3,4,5,6,7,8 };
    int mult = 3;
    size_t s = std::count_if(vec.begin(), vec.end(), isMultOf(mult));
    std::cout<< "# of mult of "<<mult<<" is: "<<s<<"\n";
}
```

Example: counting numbers who's divided by x (Cont…)

```
//C++11 style:
//we don't need the whole struct. we need to declare a lambda
//and sent to it the mult value

void test11(){
   std::vector<int> vec{ 1,2,3,4,5,6,7,8 };
    int mult = 3;

    size_t s = std::count_if(vec.begin(), vec.end(),
            [&](int x){return !(x%mult);});

    std::cout<< "# of mult of "<<mult<<" is: "<<s<<"\n";
}
```

Example: capture section [=] and [&]

```
#include <iostream>

struct A{
    A() { std::cout<<"inside A()\n"; }
    void print() { std::cout << "printing A\n"; }
    void print() const { std::cout << "printing const A\n"; }
};

int main(){
    A a, b;
    auto fn1 = [=](){ a.print(); };
    fn1();
    auto fn2 = [&](){ a.print(); };
    fn2();
    // auto fn3 = [a](){ b.print(); }   //error. capturing only a and not b
}
```

Example: capture section [=] and [&] (Cont…)

```
int main(){
    int a, b;
    a = 1; b = 2;
    auto lambda_1 = [=] { std::cout << a << " " << b << std::endl; };
    auto lambda_2 = [&] { std::cout << a << " " << b << std::endl; };
    lambda_1(); // Prints: 1 2
    lambda_2(); // Prints: 1 2
    a = 4; b = 5;
    lambda_1(); // Prints: 1 2
    lambda_2(); // Prints: 4 5
}
```

Hands-On Exercise

- Write a code that implements a sum of all positive numbers in a range. Write it using function object and using lambda function.

- Use the file **lambda_capture_exercise.cpp** from git.

```
int main()
{
    std::vector<int> vec{ 1,2,3,4,5,6,7,8 };
    int sum = 0;
    std::for_each(vec.begin(), vec.end(), ??? );
    std::cout<< "sum of pos numbers is: "<<sum<<"\n";
}
```

Lambda Capture List

[ ] ( ) { }	no captures
[=] ( ) { }	captures everything by copy(not recommended)
[&] ( ) { }	captures everything by reference(not recommendded)
[x] ( ) { }	captures x by copy
[&x] ( ) { }	captures x by reference
[&, x] ( ) { }	captures x by copy, everything else by reference
[=, &x] ( ) { }	captures x by reference, everything else by copy

Capture Member Variable in Lambda or This Pointer

Immediately Invoked Function Expression

```
class Example{
public:
    Example() : _var{10} {}
    void func(){
        [=]() { std::cout << "inside [=]: " << _var << std::endl; }();
        [&]() { std::cout << "inside [&]: " << _var << std::endl; }();
        [this]() {std::cout<< "inside [this]: " << _var <<std::endl; }();
    }
private:
    int _var;
};

int main(){
    Example e;
    e.func();
}
```

what is the output of the following code?

```
int main() {
    int a = 4;
    auto increase_local_a = [a]() mutable {
        std::cout << a << std::endl;
        a++;
    };
    auto decrease_ref_a = [&a]() mutable {
        std::cout << a << std::endl;
        a--;
    };
    increase_local_a();      //(1)
    decrease_ref_a();        //(2)
    increase_local_a();      //(3)
    decrease_ref_a();        //(4)
    increase_local_a();      //(5)
    decrease_ref_a();        //(6)
    std::cout << a << std::endl; //(7)
}
```

what is the output of the following code? (Cont…)

```
int main() {
    int a = 4;
    auto increase_local_a = [a]() mutable {
        std::cout << a << std::endl;
        a++;
    };
    auto decrease_ref_a = [&a]() mutable {
        std::cout << a << std::endl;
        a--;
    };
    increase_local_a();      // (1) Prints: 4
    decrease_ref_a();        // (2) Prints: 4
    increase_local_a();      // (3) Prints: 5
    decrease_ref_a();        // (4) Prints: 3
    increase_local_a();      // (5) Prints: 6
    decrease_ref_a();        // (6) Prints: 2
    std::cout << a << std::endl; // (7) Prints: 1
}
```

mutable lambda

- lambda's operator() is const by-default, meaning it can't modify the variables it captured by-value (which are analogous to class members). To change this default, add mutable:

```
int i = 1;
[&i](){ i = 1; }; // ok, 'i' is captured by-reference.
[i](){ i = 1; }; // ERROR: assignment of read-only variable 'i'.
[i]() mutable { i = 1; }; // ok.
```

- This gets even more interesting when talking about copying lambdas. Key thing to remember - they behave like classes:

```
int i = 0;
auto x = [i]() mutable { std::cout << ++i << std::endl; }
x();    //output: 1
auto y = x;
x();    //output: 2
y();    //output: 2
```

Lambda's size

- Because lambdas have captures, there's no single size for all lambdas. Example:

```
auto f1 = [](){};
std::cout << sizeof(f1) << std::endl;   //output: 1

std::array<char, 100> ar;
auto f2 = [&ar](){};
std::cout << sizeof(f2) << std::endl;   //output: 8 (64 bit)

auto f3 = [ar](){};
std::cout << sizeof(f3) << std::endl;   //output: 100
```

https://shaharmike.com/cpp/lambdas-and-functions/#mutable-lambdas

C++14: template Lambda Expressions

- **C++11 problem:** lambdas can't be templated, but `decltype()` seems to help when using a lambda within a templated class or function.

```
template<typename T>
void func(T t){
    //declaring a lambda who’s parameter is of type T
    auto fn = [](decltype(t) t){ return t;};
    //invoke the lambda with parameter t
    std::cout << fn(t) << std::endl;
}

int main() {
    std::string s("My string");
    func(s);
    func(1024);
    func(true);
}
```

- **C++14 solution**: use the `auto` keyword to make the lambda generic.
  Now the lambda-function can automatically deduce the types of its arguments and return type.

```
auto max = [](auto a, auto b) {return a>b?a:b;};

 std::cout<< max(1,2) <<std::endl;
 std::cout<< max(1.5,2) <<std::endl;
 std::cout<< max(1,2.8) <<std::endl;
```

Generalized Lambda Capture

- **C++11 problem**: In order to catch a parameter, we can use by-value/by reference but the parameter is in the whole function's scope.
  What if we want it to be used only inside the lambda?

```
int main(){
    std::ofstream out{"output.txt"};
    auto write = [&out](int x){
        out<< x <<std::endl;
    };
    //we don't wat out to be used here!
}
```

Generalized Lambda Capture (Cont…)

- Another problem: `std::unique_ptr`. In C++11 It can be captured only by reference since it is no copyable. How can we send it with passing the control to the function?

```
int main(){
    std::unique_ptr<A> ptr(new A);

    auto fn = [ptr](){ ptr->print(); return *ptr; };
            //Error! ptr has no copy constructor!
    fn();
}
```

C++14: Lambda capture expressions (Cont…)

- **C++14 solution**: C++14 allows captured members to be initialized with arbitrary expressions. This allows both capture by value-move and declaring arbitrary members of the lambda, without having a correspondingly named variable in an outer scope.

```
int main(){
    int data{5};
    auto f = [y = data](int arg){ return y + arg;}(data);
}
```

C++14: Lambda capture expressions (Cont…)

- Now we can use std::move to transfer the ownership of the unique_ptr to the lambda:

```
int main(){
    std::unique_ptr<A> ptr(new A);
    // auto fn = [ptr](){ ptr->print(); return *ptr; };
                //Error! ptr has no copy constructor!

    auto fn = [p = std::move(ptr)](){ p->print(); return *p; };
}
```

C++17: lambda capture of *this

- **C++14 problem**: we can capture this in many ways since C++11, but if we want to capture a copy of the object and not the original one, in C++14 we need to write:

```
[tmp = *this]() {std::cout<<"inside [temp=*this]: " <<
                 tmp._var <<std::endl; }(); //C++14
```

- **C++17 solution**: we can simply capture [*this].

*C++17: lambda capture of *this (Cont…)*

```
class Example{
public:
    Example() : _var{10} {}
    void func()
    {
        [*this]() mutable { ++_var;
                            std::cout<< "inside [*this]: " << _var <<std::endl;
                          }();
    }
public:
    int _var;
};

int main(){
    Example e;
    e.func();
    std::cout<<e._var<<std::endl;
}
```

*C++17: lambda capture of *this (Cont…)*

- Suppose we have the following member function:

```
std::string getDescription(const std::string& header){
    std::string desc = header;
    desc += "\n";
    desc = desc + "name: " + _name + "\n";
    desc = desc + "id: " + std::to_string(_id) + "\n";
    return desc;
}
```

*C++17: lambda capture of *this (Cont…)*

- in the main:

```
int main(){
    Data d("product",10);
    auto desc = d.getDescription("=====");
    std::cout<<desc<<std::endl;
}
```

- What if we do not want to print desc? We paid for creating the string on the heap and copying it.

*C++17: lambda capture of *this (Cont…)*

- We can use lambda and call it only if we want to print the result:

```
auto getDescription(){
    return [this](const std::string& header){
        std::string desc = header;
        desc += "\n";
        desc = desc + "name: " + _name + "\n";
        desc = desc + "id: " + std::to_string(_id) + "\n";
        return desc;
    };
}
```

*C++17: lambda capture of *this (Cont…)*

- It won't work if the object is on the heap since we store the lambda and while we call it the object is already deleted…
- **C++17** allows us to send [*this] instead of just [this].

constexpr lambda

- If the lambda expression satisfies the conditions of `constexpr` expression, it will automatically become `constexpre`, without the need to specify it.

```
int main(){

    auto f = [](int x, int y){
        return x+y;
    };
    std::cout<<f(2,3)<<std::endl;
}
```

https://godbolt.org/z/eG3ad3

std::function

std::function

- Class template `std::function` is a general-purpose polymorphic function wrapper.

- Instances of `std::function` can store, copy, and invoke any `CopyConstructible` Callable target -- functions, lambda expressions, bind expressions, or other function objects, as well as pointers to member functions and pointers to data members.
  - The stored callable object is called the target of `std::function`.
  - If a `std::function` contains no target, it is called empty.
  - Invoking the target of an empty `std::function` results in `std::bad_function_call` exception being thrown.

std::function (Cont…)

- The signature of std::function is defined as follows:

```
std::function < return_type (parameter0, parameter1, ...) >
```

- For example: a function that represents void f() is:

```
auto func = std::function<void()>{};
```

- It can be initialized with function pointer, function object or a lambda.

std::function (Cont…)

```
#include <iostream>
#include <functional>

int main(){
    auto func = std::function<void(int)>{
        [](int v)
        { std::cout << "value is " << v << std::endl; }
    };
    func(12);
}
```

```
class ThreadPool {
public:
    ThreadPool(size_t);

    template<class F, class... Args>
    auto addTask(F&& f, Args&&... args)
        -> std::future<typename std::result_of<F(Args...)>::type>;

    ~ThreadPool();
private:
    void createThreads(size_t);

private:
    std::vector< std::thread > _workers;
    std::queue< std::function<void()> > _tasks;

    // synchronization
    std::mutex _queue_mutex;
    std::condition_variable _condition;
    bool _stop;
};
```

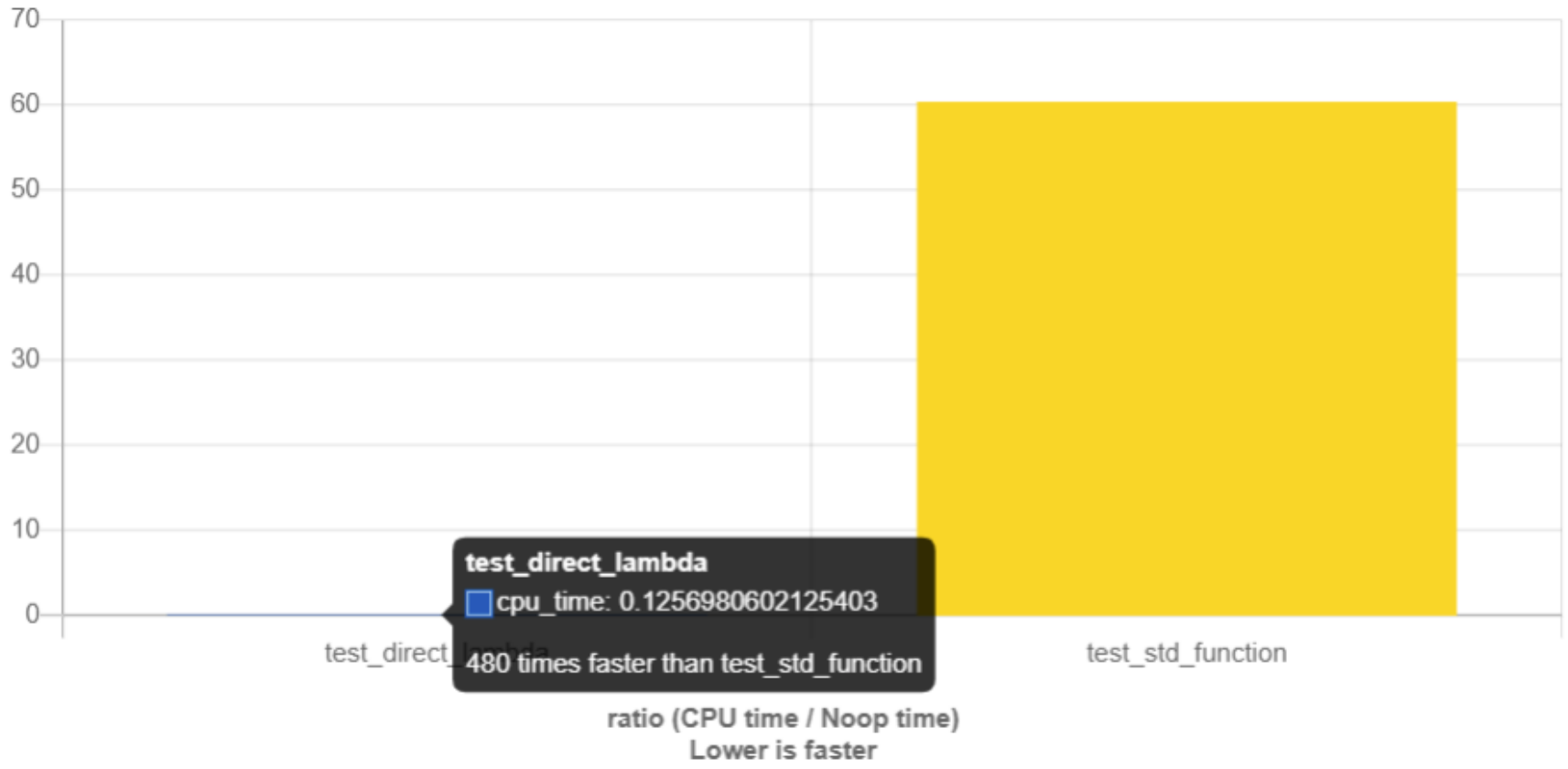
std::function (Cont…)

- A std::function has a few performance losses compared to a lambda function:
  - `std::function` **cannot be inlined**. The compiler inlines the lambda call.
  - `std::function` **heap allocates and captures variables**. If a lambda function with captured variables/references is assigned to a `std::function`, the `std::function` will, in most cases, allocate space on the heap to store the captured variables (if it is bigger than a threshold size).
  - **Invoking an `std::function` requires a few more operations than a lambda**.

compiler = Clang 13.0 | std = c++20 | optim = O3 | STL = libstdc++(GNU)

Run Benchmark | Record disassembly | Clear cached results

Charts | Assembly



std::bind

- std::bind is a Standard Function Objects that acts as a Functional Adaptor i.e. it takes a function as input and returns a new function Object as an output with one or more of the arguments of passed function bound or rearranged.

- std::bind takes a function as its first parameter and then that function’s argument as its parameter.

std::bind (Cont…)

```
#include <functional>
#include <iostream>

int add(int first, int second)
{
    return first + second;
}

int main(){
    using namespace std::placeholders;  // for _1, _2, _3...
    auto add_func = std::bind(&add, _1, _2);
    std::cout<<"result = "<<add_func(4,5)<<std::endl;
}
```

std::bind (Cont…)

```
int main(){
    using namespace std::placeholders;  // for _1, _2, _3...
    std::array<int,10> arr{1,20,13,4,5,6,10,28,19,15};

    int divided_by = 5;
    auto lbd = [&divided_by](int num){return num%divided_by==0;};
    std::cout << std::count_if(std::begin(arr), std::end(arr) ,
                               std::bind(lbd, _1));
}
```

variadic templates

Variadic Templates

- **C++98 problem**: writing a function that takes an arbitrary number of arguments is not safe, cannot be templated and done at runtime.
- Writing a printf-like function with (…) is not strong typed and uses low-level memory manipulations and often causes seg-faults.
- Templates (classes and functions) could only take a fixed number of arguments, which had to be specified when a template was first declared.

Variadic Templates (Cont…)

- Using `std::initializer_list` can be a good solution of all the arguments are of the same type:

```
template<typename T>
void print(std::initializer_list<T> args){
    for(const auto &x : args){
        std::cout<< x <<" ";
    }
    std::cout<<std::endl;
}

int main(){
    print({1,2,3,4,5.5});   //ERROR!
}
```

Variadic Templates (Cont…)

- **C++11 solution**: **variadic template**.
- Variadic templates are template that take a variable number of arguments. Variadic function templates are functions which can take multiple number of arguments.
- In this function, we cannot access the arguments directly…

```
template<typename... Params>
void print2(Params...args){
    //we cannot access elements directly
}
```

Variadic Templates (Cont…)

- We use `typename... Ts` to declare Ts as a so-called **template parameter pack**. The ellipsis (...) operator is used here to declare that Ts truly is a template parameter pack.

- The function signature accepts a closely-related **function parameter pack** – in other words, a bag of parameters, whose types are given by the aforementioned template parameter pack.

Variadic Templates (Cont…)

- We need to use recursion in order to access the arguments:

```
void print2(){
    std::cout<<std::endl;
}

template<typename T, typename... Params>
void print2(T a, Params...args){
    //we cannot access elements directly
    //we need to use recursion
    std::cout<<a<<" ";
    print2(args...);
}

int main(){
    print2(1,2,3,4,5.5);
}
```

Variadic Templates (Cont…)

- `sizeof…` is the size of the parameters pack

```
template<typename T, typename... Params>
void print2(T a, Params...args){
    // std::cout<<sizeof...(args)<<std::endl;
    std::cout<<a;
    if(sizeof...(args)!=0){
        std::cout<<", ";
    }
    print2(args...);
}
```

Variadic Templates (Cont…)

- Instead of sending the parameters by value, we can send them by reference:

```
void print3(){
    std::cout<<std::endl;
}

template<typename T, typename... Params>
void print3(const T& a, const Params&...args){
    std::cout<<a;
    if(sizeof...(args)!=0){
        std::cout<<", ";
    }
    print3(args...);
}
```

- If we send r-value reference to this print function, we get inside the function a l-value type, so sending it to the recursion will not use the move semantics.
  We can fix it:

```
template<typename T, typename... Params>
void print4(T&& a, Params&&...args){
    // std::cout<<sizeof...(args)<<std::endl;
    std::cout<<a;
    if(sizeof...(args)!=0){
        std::cout<<", ";
    }
    print4(std::forward<Params>(args)...);
}
```

Variadic Templates (Cont…)

- Variadic template is very useful for a logger. Instead of writing too many methods, we can simply send whatever we want to print, the work will be done!

```
void log(){ /*...*/ }

template<typename T, typename ... Args>
void log(T first, Args ... args) {
    // Print the First Element
    std::cout<<first<<" , ";
    // Forward the remaining arguments
    log(args ...);
}
```

Variadic Templates: Hands-On Exercise

- Implement a **factory method** that creates an instance of some type T, initializes it with arguments and returns it.

```
int main(){
    std::shared_ptr<int> p1 = create<int>(5);

    std::shared_ptr<std::string> s = create<std::string>("hello");

    std::shared_ptr<Employee> e = create<Employee>("Bob", 324567, 10000);

    std::shared_ptr<Contact> c = create<Contact>("Joe","0503987342");
}
```

C++17 Fold Expressions

C++17 Fold Expressions

- **Folding** is a process that applies a binary operator to a list of values recursively: (1 + (2 + (3 + (4 + (5)))))
- It is also used when calculating a variadic template function.

- **C++11 problem:** variadic templates required some additional code when you wanted to implement 'recursive' functions like sum, all, etc’.

Fold Expressions (Cont…)

- **C++17 solution:** C++17 offers a new way to unpack variadic parameters with binary operators.

- Syntax:

```
(args  op ...)
(...   op args)
(args  op ... op ... init)
(init  op ... op ... args)
```

Fold Expressions (Cont…)

```
//C++17 Fold Expression:
template<typename...Args>
auto sum17_right_fold(Args...args){
    return (args + ...);
}

template<typename...Args>
auto sum17_left_fold(Args...args){
    return (... + args);
}
```

Fold Expressions (Cont…)

- In case the function sum is called without any arguments, the variadic parameter pack contains no values that could be folded.
- For most operations this is an error.
- We need to decide whether this should stay as an error, or if an empty sum should result in a specific value, like 0.

Fold Expressions (Cont…)

```
template<typename...Args>
auto sum(Args...args)
{
    return (args + ... + 0);
}

int main()
{
    result = sum();
    std::cout<<"result is: "<<result<<std::endl;
}
```

Fold Expressions (Cont…)

- Here is a nice implementation of a printf function using folds:

```
template<typename ...Args>
void foldPrint(Args&&... args) {
    (std::cout <<  ... <<  std::forward<Args>(args)) << std::endl;
}

int main()
{
    int x = 8;
    foldPrint("hello", ", ", 10, ", ", 90.0, ", ", x);
}
```

Fold Expressions (Cont…)

EXPRESSION	EXPANSION
(... op pack)	((pack1 op pack2) op ...) op packN
(init op ... op pack)	(((init op pack1) op pack2) op ...) op packN
(pack op ...)	pack1 op (... op (packN-1 op packN))
(pack op ... op init)	pack1 op (... op (packN-1 op (packN op init)))

Also by default we get the following values for empty parameter packs:

OPERATOR	DEFAULT VALUE
&&	true
||	false
,	void()

Fold Expressions (Cont…)

```
template<typename...Args>
bool isAllPositive(Args...args){
    return ((args>0)&&...);
}
int main()
{
    std::cout<<"are all positive? "<<std::boolalpha
                    << isAllPositive(1,2,3,5,4)<<std::endl;
    std::cout<<"are all positive? "<<std::boolalpha
                    << isAllPositive(1,2,-3,5,-4)<<std::endl;
}
```

Expressions Fold – Hands On Exercise

- Implement the function sumBetween which gets the following parameters:
  - begin
  - end
  - elements to sum

- The function returns the sum of all elements that are in the range [begin,end].

```
auto res = sumBetween(2,5,1,2,3,4,5,6,7,8,9);
std::cout<<"result is: "<<res<<std::endl;   //output: 14
```

noexcept operator

noexcept operator

- The **noexcept operator** performs a compile-time check that returns true if an expression is declared to not throw any exceptions.

- It can be used within a function template's **noexcept** specifier to declare that the function will throw exceptions for some types but not others.

Exceptions Recap

- Exceptions is a mechanism for handling runtime errors.
- Do not let the user to ignore error codes.
- Throwing an exception causes the current function to popped out of the call stack, thus call destructors of local stack-based objects.
- Exceptions can be nested and re-thrown again.
- We should not throw an exception from a destructor.

Exception specifications

- We wish to know if a certain function may throw an exception, and what is the type of this exception.

- An **exception specification** provides a solution that can be used to list the exceptions a function may throw with the function declaration.
- It is supposed to guarantee that the function does not throw any other types of exceptions, but most compilers does not support it…

noexcept operator (cont...)

- An exception declaration is part of the function's interface, and it must be specified on the function declarations that appear in header files.

```
class Stack {
public:
   // ...
   void pop( int &value ) throw(popOnEmptyException);
   void push( int value ) throw(pushOnFullException);
private:
   // ...
};
```

noexcept operator (cont...)

- Violations of a function's exception specification are detected only at **run-time**.
  - The compiler does not generate compile-time errors if an expression can throw an exception of a type disallowed by the exception specification.

```
struct ExceptionType{};

void processData( int, int ) throw(std::string, ExceptionType){
    throw ExceptionType();
}

void doAction (int op1, int op2 ) throw(std::string) {
    processData( op1, op2 ); // no compile-time error - but get runtime error!
   // ...
}
```

noexcept operator (cont...)

- An empty exception specification guarantees that the function does not throw any exception.

```
void no_problem() throw();
```

- If a function declaration does not specify an exception specification, the function can throw exceptions of any type.
- No type conversion is allowed between the type of the exception thrown and a type specified by the exception specification.

noexcept operator (cont…)

- **C++98 problem**: the feature of exception specification doesn't really work… some of the compilers even ignores it…

- In addition, there is a runtime penalty since there are library methods that would prefer to use methods that do not throw exceptions, over neutral exception ones, and it is not clear in C++98 which function is what.

noexcept operator (cont...)

- In **C++11 standard**, the compiler only wants to know whether to generate the unwinding code for this function or not. The type of the exception thrown is the problem of the programmer.

- This is done by adding the Boolean operator `noexcept(condition)` to each function.

noexcept operator (cont...)

- `noexcept(true)` – the function does not throw any exception; thus no unwinding code is generated.
- Simply writing `noexcept` is automatically converted to `noexcept(true)` by the compiler.
- In C++11 all destructors are automatically declared as `noexcept(true)` since they should not throw any exception.

```
void a() {
    throw std::string("from a");
}

void b(){
    a();
}

void c() noexcept {
    b();
}

int main(){
    // c(); //will cause runtime error of std::terminate()

    std::cout<<std::boolalpha<<noexcept(c())<<std::endl; //true
    std::cout<<std::boolalpha<<noexcept(b())<<std::endl; //false
}
```

noexcept operator (cont...)

- A function can declare that it has the same `noexcept` specification as the function is calls:

```
void test1(int x){
    throw x;
      //function is implicitly declared as noexcept(false) since it throws
}

void test2(int x) noexcept(true){}

int someFunction(int x) noexcept(noexcept(test1(x))) {
    //someFunction now has the same noexcept specification as test1
    test1(x);

     return x;
}
```

noexcept operator (cont...)

- There is a difference between declaring the function as `throw()` and `noexcept(true)`.

```
int f(int x) throw();   // no exceptions from f: C++98 style

int f(int x) noexcept;  // no exceptions from f: C++11 style
```

noexcept operator (cont...)

- **In the C++98 version** – if `f()` is throwing an exception at runtime, the call stack is unwound to f's caller and may cause the program to terminate.

- **In the C++11 version**, if `f()` is throwing an exception at runtime, the stack is only **possibly** unwound before program execution is terminated.

- So, what?...

noexcept operator (cont...)

- **The difference** between `noexcept` and `throw()` is that **in case of `throw()`** the exception stack is still unwound, and destructors are called, so implementation must keep track of the stack, since it calls `std::unexpected`,
  while **`constexpt`** calls directly to `std::terminate`, which does not require the stack to be unwound (it is implementation defined).

- From StackOverflow:
  - GCC seem to really not unwind the stack for noexcept: Demo
  - While clang still unwinds: Demo

noexcept operator (cont...)

- A function that calls a library function, cannot guarantee that it will never propagate an exception, thus should be declared as `noexcept(false)`.

```
void test() noexcept(false){
    std::cout<<"hello"<<std::endl;
    //since test3 is calling a library function,
    //it cannot be sure that it will now
    //throw an exception
}
```

C++17 changes in noexcept operator

```
void func1() noexcept{}

void func2(){ throw ("hello"); }

int main(){
   //until C++17: the noexcept isn’t part of the function's signature
   void(*p)() noexcept;
    p = func2;
   p();    //can throw
   //compiling with -std=c++17 will cause a compilation error
}

void func3() throw(int){
     //exception specifications is deprecated until C++17
     //and removed in C++17
   throw 5;
   //compiling with -std=c++17 will cause an error instead of warning
}
```

Attributes

Attributes

- Compilers can get attributes inside the code that helps them to produce and optimize the code.

```
struct S { short f[3]; } __attribute__ ((aligned (8)));

void fatal () __attribute__ ((noreturn));

#if COMPILING_DLL
    #define DLLEXPORT __declspec(dllexport)
#else
    #define DLLEXPORT __declspec(dllimport)
#endif
```

- In C++ that meta information is fixed by the compiler, you cannot add user-defined attributes.

Attributes (Cont…)

- With the modern C++, we get more and more standardized attributes that will work with other compilers.
- We're moving a bit from compiler specific annotation to standard forms.

- In C++11/14 added a nicer form of specifying annotations over the code.
- Syntax: `[[attr]]` or `[[namespace::attr]]`

Attributes (Cont…)

- C++11 attributes:
  - `[[noreturn]]` - for example [[noreturn]] void terminate() noexcept;
  - `[[carries_dependency]]` - mostly to help optimizing multi-threaded code and when using different memory models

- C++14 attributes:
  - `[[deprecated]]` and `[[deprecated("reason")]]`

Attributes (Cont…)

```
[[deprecated("[FUNC DEP] this version is not supported anymore")]]
void oldFunction(){
}

class [[deprecated("[CLASS DEP] this class is not supported anymore")]]
MyClass{
};

namespace [[deprecated("[NAMESPACE DEP] this namespace is not supported anymore")]]
experimental_library
{
    void func(){}
}
```

Attributes (Cont…)

- C++17 attributes:
  - `[[fallthrough]]` - Indicates that a fall-through in a switch statement is intentional and a warning should not be issued for it.
  - `[[nodiscard]] -` used to stress that the return value of a function is not to be discarded, on pain of a compiler warning. This attribute can also be applied to types in order to mark all functions which return that type as [[nodiscard]].
  - `[[maybe_unused]]` - Suppresses compiler warnings about unused entities when they are declared with [[maybe_unused]].

Attributes (Cont…)

```
void test_fallthrough()
{
    char c = getChar();

    switch (c)
    {
    case 'a':
        f(); // Warning! fallthrough is perhaps a programmer error
    case 'b':
        g();
    [[fallthrough]]; // Warning suppressed, fallthrough is ok
    case 'c':
        h();
    }
}
```

Attributes (Cont…)

```
// [[nodiscard]] int foo() {return 5;}  //warning!
int foo(){return 5;}    //no warning

void test_nodiscard_1() {
    foo(); // Warning! return value of a
           // nodiscard function is discarded
}

[[nodiscard]] struct DoNotThrowMeAway{};
DoNotThrowMeAway func();
void test_nodiscard_2() {
    func();     // Warning emitted, return value of a
                // nodiscard function is discarded
}
```

Attributes (Cont…)

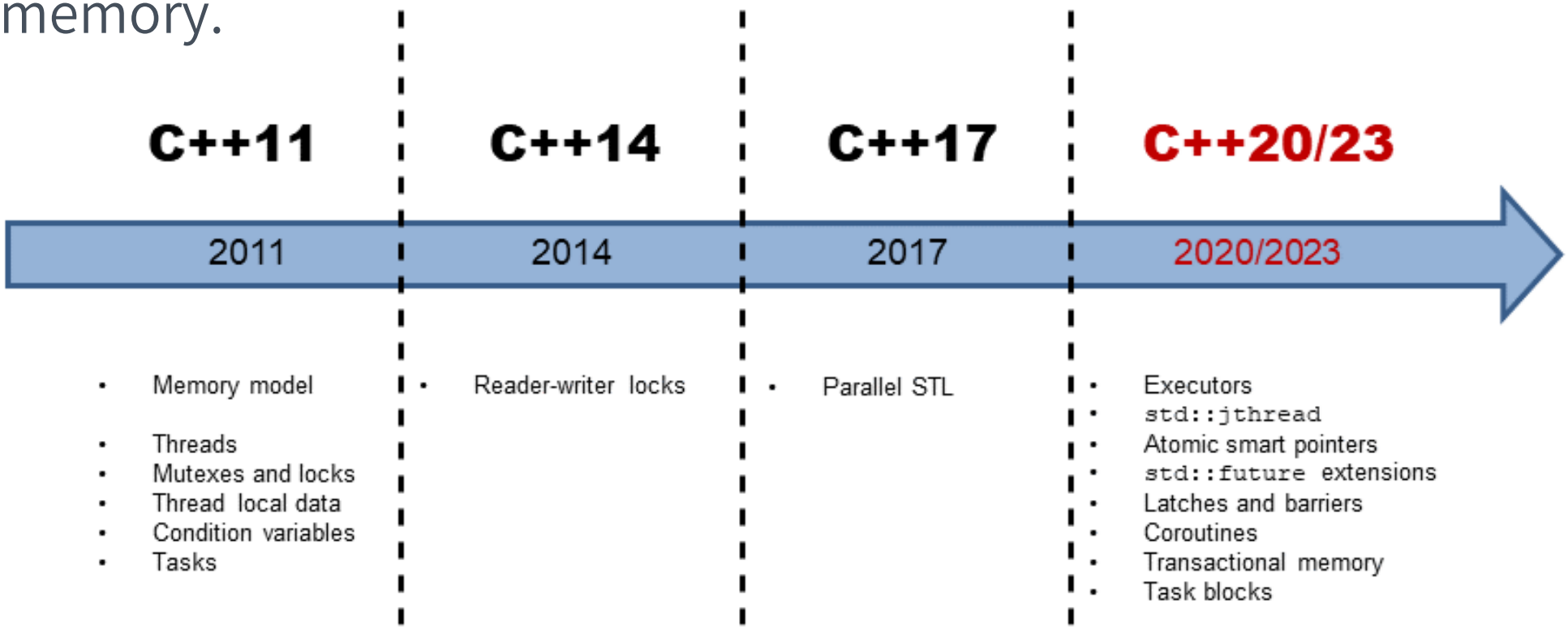
```
static void impl1() { /*...*/ } // Compilers may warn about this
[[maybe_unused]] static void impl2() { /*...*/ }
                // Warning suppressed


void test_maybe_unused() {
   int x = 42; // Compilers may warn about this
   [[maybe_unused]] int y = 42; // Warning suppressed
}
```

C++ Memory Consistency Model

C++ Memory Consistency Model

- The **memory consistency model** is a specification of the allowed behavior of multithreading programs executing with shared memory.



C++ Memory Consistency Model (Cont…)

- **Pre-C++11** had no mention of threads in the standard. Everything was implementation-dependent.

```
if (x >= 0 && x < 3) {
    switch (x) {
    case 0: do0(); break;
    case 1: do1(); break;
    case 2: do2(); break;
    }
}
```

If x is a shared data and the compiler optimizes the code to one jump statement, can we be sure that it is safe?

C++ Memory Consistency Model (Cont…)

- The CPU also has a memory model of its own.
  - In addition, can perform optimizations such as loose cache coherence protocols and out-of-order execution etc’.
- The compiler can have optimizations:
  - Reordering to hide load latencies.
  - Using registers to avoid repeating loads ad stores.
- **But since threads are part of the language since C++11, their behavior must be fully defined, to gain portability and simplicity to the programmer.**

MT Support in Modern C++

- **C++98 problem**: no standard for multi-threading library.
  - There are some wrappers (boost, ACE, etc'). But no standard.

- **C++11 solution**: set rules that different threads need to follow when sharing data in memory.

C++ Memory Consistency Model (Cont…)

COMPILER EXPLORER | Add... | More | Listen to the Cpp.chat podcast | Sponsors intel PC-lint Solid Sands | Share | Other

C++ source #1 — C++

```cpp
//shared string
int messageToPublish;
int rawMessage;
int readyToPublish;

//the producer
void publishMessage()
{
    messageToPublish = rawMessage;
    rawMessage = 0;
    readyToPublish = 1;
}
```

x86-64 gcc 10.2 (Editor #1, Compiler #1) C++ — x86-64 gcc 10.2 — Compiler options...

Output... | Filter... | Libraries | Add new... | Add tool...

```asm
messageToPublish:
        .zero   4
rawMessage:
        .zero   4
readyToPublish:
        .zero   4
publishMessage():
        push    rbp
        mov     rbp, rsp
        mov     eax, DWORD PTR rawMessage[rip]
        mov     DWORD PTR messageToPublish[rip], eax
        mov     DWORD PTR rawMessage[rip], 0
        mov     DWORD PTR readyToPublish[rip], 1
        nop
        pop     rbp
        ret
```

COMPILER EXPLORER https://compiler-explorer.com/z/cxzKrGc1q

C++ Memory Consistency Model (Cont…)

COMPILER EXPLORER

C++ source #1

```cpp
//shared string
int messageToPublish;
int rawMessage;
int readyToPublish;

//the producer
void publishMessage()
{
    messageToPublish = rawMessage;
    rawMessage = 0;
    readyToPublish = 1;
}
```

x86-64 gcc 10.2 (Editor #1, Compiler #1)

-O3

```asm
publishMessage:
        mov     DWORD PTR readyToPublish[rip], 1
        mov     eax, DWORD PTR rawMessage[rip]
        mov     DWORD PTR rawMessage[rip], 0
        mov     DWORD PTR messageToPublish[rip], eax
        ret
readyToPublish:
        .zero   4
rawMessage:
        .zero   4
messageToPublish:
        .zero   4
```

Memory Barriers

- In order to synchronize data between two threads, we need that:
  - No torn reads or writes
  - Threads can see the latest values
- The memory barriers prevent reordering of memory operations:
  - At compile time (the compiler)
  - At runtime (the CPU)
- **C++11** supports `std::atomic` and `std::memory_order` new libraries for achieving low-level memory management.

YouTube

CppCon 2017: Fedor Pikus **"C++ atomics, from basic to advanced. What do they really do?"**

The <atomic> Library

- **Atomic types** are **templated types** that encapsulate a value whose access is guaranteed to not cause data races and can be used to synchronize memory accesses among different threads.
  - That is: if one thread writes to an atomic object while another thread reads from it, the behavior is well-defined.

- This header declares two C++ classes, `atomic` and `atomic_flag`, that implement all the features of atomic types in self-contained classes.

The <atomic> Library (Cont…)

- **The purpose:**
  - hold a memory data that all other threads will always see its final state.
  - Be able to change its value as an atomic operation.
  - Be able to do if-statement on it as an atomic operation.

- Syntax: `atomic<type> var;`
- Header file: `<atomic>`

The <atomic> Library (Cont…)

- `std::atomic` provides `store(T)` and `load()` functions that atomically set and get the contents of the std::atomic.
- The function `exchange` sets the atomic to a new value and returns the value held previously.
- The functions `compare_exchange_weak` and `compare_exchance_strong` performs atomic exchange only if the value is equal to the provided expected value.
  - These two functions can be used to implement lock-free algorithms.

The <atomic> Library (Cont…)

Member functions

(constructor)	constructs an atomic object (public member function)
`operator=`	stores a value into an atomic object (public member function)
`is_lock_free`	checks if the atomic object is lock-free (public member function)
`store`	atomically replaces the value of the atomic object with a non-atomic argument (public member function)
`load`	atomically obtains the value of the atomic object (public member function)
`operator T`	loads a value from an atomic object (public member function)
`exchange`	atomically replaces the value of the atomic object and obtains the value held previously (public member function)
`compare_exchange_weak` `compare_exchange_strong`	atomically compares the value of the atomic object with non-atomic argument and performs atomic exchange if equal or atomic load if not (public member function)
`wait` (C++20)	blocks the thread until notified and the atomic value changes (public member function)
`notify_one` (C++20)	notifies at least one thread waiting on the atomic object (public member function)
`notify_all` (C++20)	notifies all threads blocked waiting on the atomic object (public member function)

Specialized member functions

`fetch_add`	atomically adds the argument to the value stored in the atomic object and obtains the value held previously (public member function)
`fetch_sub`	atomically subtracts the argument from the value stored in the atomic object and obtains the value held previously (public member function)
`fetch_and`	atomically performs bitwise AND between the argument and the value of the atomic object and obtains the value held previously (public member function)
`fetch_or`	atomically performs bitwise OR between the argument and the value of the atomic object and obtains the value held previously (public member function)
`fetch_xor`	atomically performs bitwise XOR between the argument and the value of the atomic object and obtains the value held previously (public member function)

`operator++`
`operator++(int)`
`operator--`
`operator--(int)`
	increments or decrements the atomic value by one
	(public member function)

`operator+=`
`operator-=`
`operator&=`
`operator|=`
`operator^=`
	adds, subtracts, or performs bitwise AND, OR, XOR with the atomic value
	(public member function)

The <atomic> Library (Cont…)

- **Note**: atomic variables are rarely used by themselves, since all variables needs to be atomic in order to get its efficiency.

- Usually, **they point to a non-atomic variable**. They are a gateway to memory access.

- **In order to make it work, we need:**
  - **Before acquiring:** the memory we refer to must be completely updated.
  - **Releasing to the shared memory:** data must become visible to everyone.

- **Use memory barriers for that!**

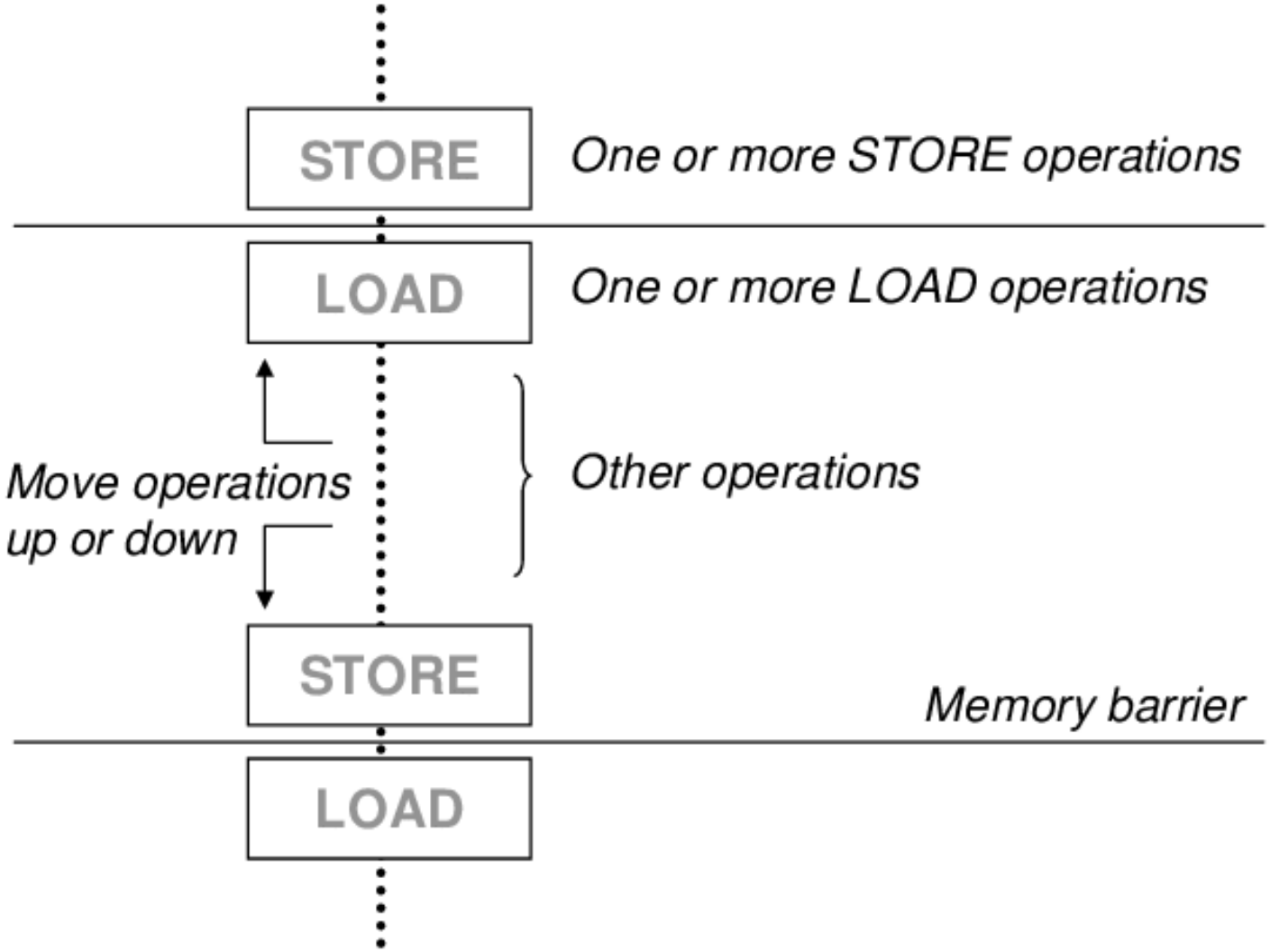
- Memory barriers control how changes to memory made by one CPU become visible to other CPUs.
  - It is global across all CPUs.
  - Implemented by the Hardware.
  - Ensures the specific order of read and writes.
  - C++11 provides standard memory barriers. They are called `std::atomic_thread_fence`, and simply called fences or memory barriers.

The <atomic> Library (Cont…)

- There are four different ways to combine load and store operations:
  - **LoadLoad**: A load followed by a load. (load = acquire)
  - **LoadStore:** A load followed by a store. (store = release)
  - **StoreLoad:** A store followed by a load.
  - **StoreStore:** A store followed by a store.

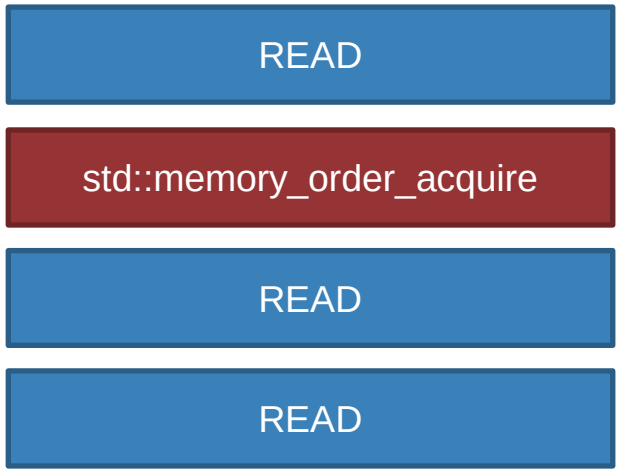
- In case you place memory barriers between two operations like **LoadLoad**, **LoadStore**, **StoreLoad** or **StoreStore**, you have the guarantee, that specific **LoadLoad**, **LoadStore**, **StoreLoad** or **StoreStore** operations can not be reordered.

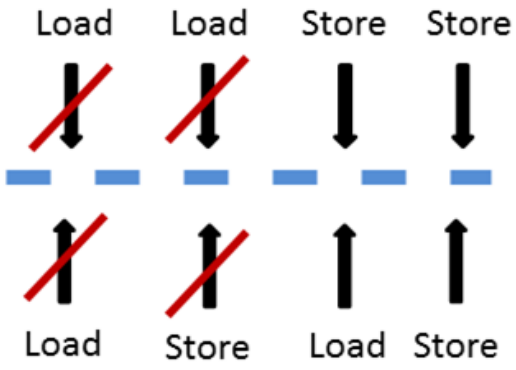
The <atomic> Library (Cont…)



std::memory_order

- **`std:memory_order_relaxed`** – no memory barrier. We can switch read and write operations as we wish. No guarantees.
- **`std:memory_order_acquire`** – also called “half barrier”. It guarantees that all memory operations scheduled after the barrier in the program order become visible after the barrier.

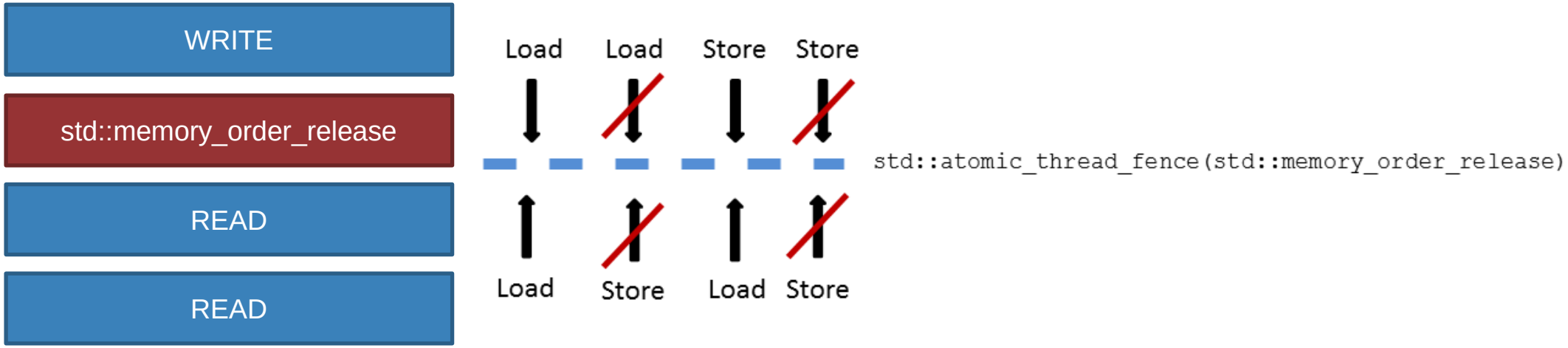





```
std::atomic_thread_fence(std::memory_order_acquire)
```

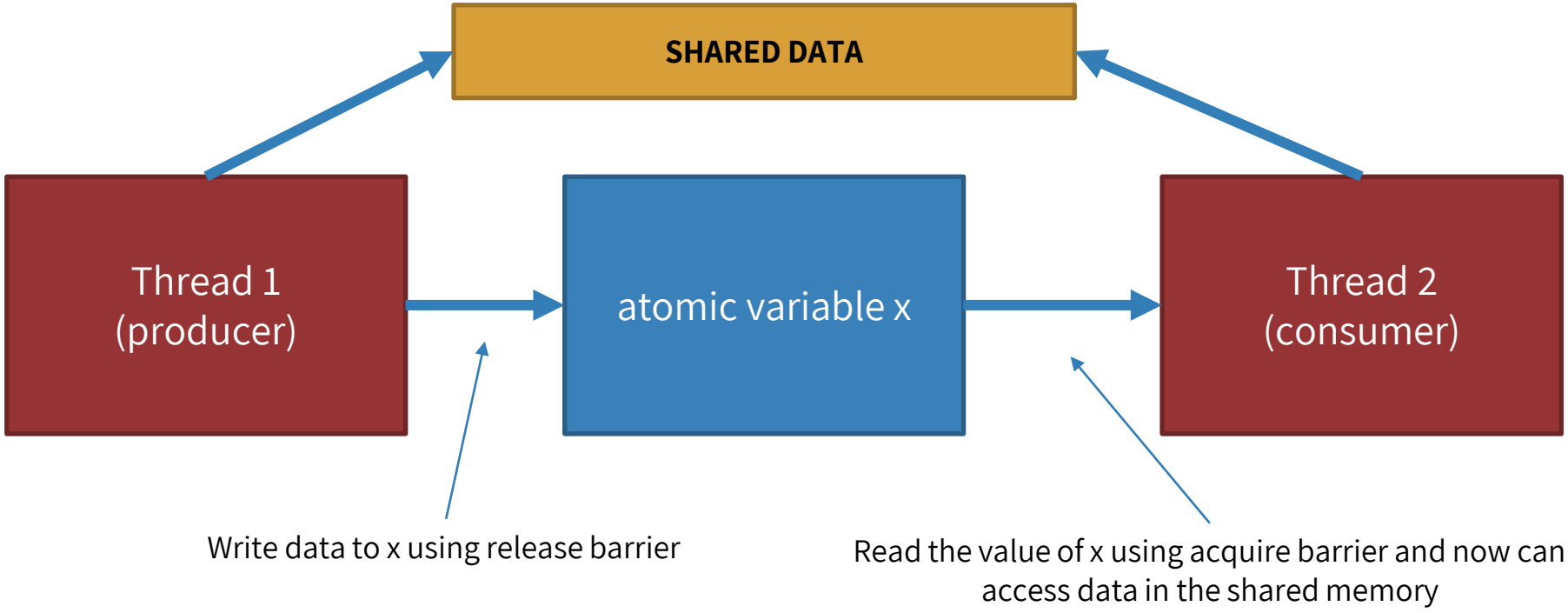
std::memory_order (Cont…)

- `std:memory_order_release` - prevents that a read or write operation before a release fence can be reordered with a write operation after a release fence.



std::memory_order (Cont…)

- In order this to work – **both threads need to use barriers from both sides**: the one that updates the variable (release) and those who wish to read it (acquire).



std::memory_order (Cont…)

reference to the value expected to be found in the atomic object.

the value to store in the atomic object if it is as expected

- `bool std:compare_exchange_strong( T& expected, T desired, std::memory_order success, std::memory_order failure )`

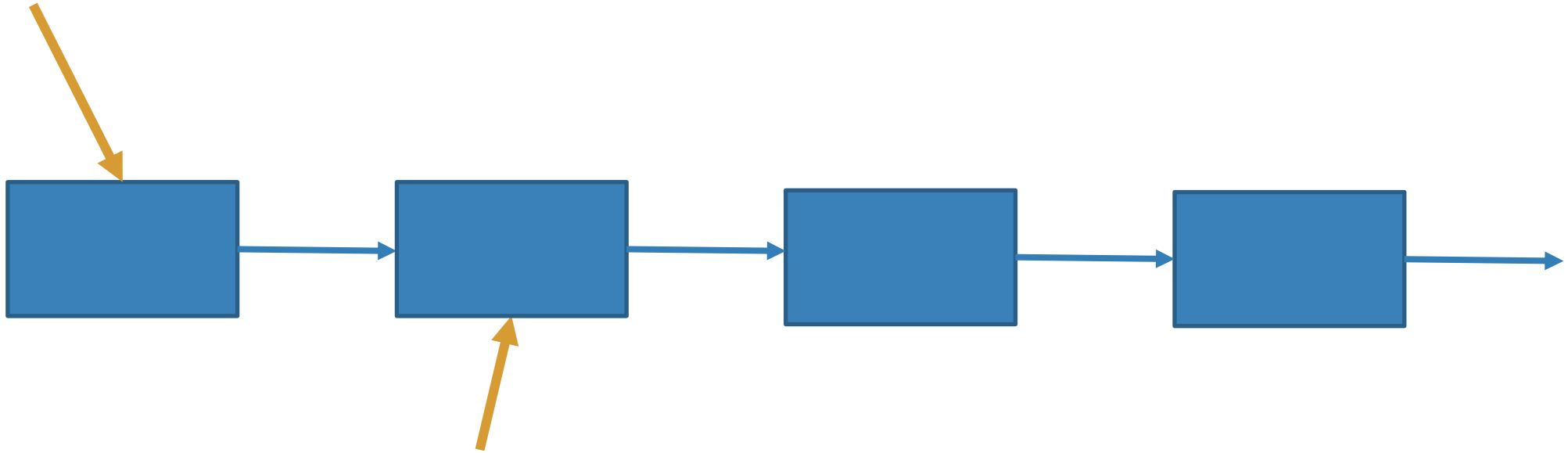
true if the underlying atomic value was successfully changed, false otherwise.

the memory synchronization ordering for the load operation if the comparison fails.

atomically compares the object representation of *this with that of expected:
if those are bitwise-equal, replaces the former with desired (performs read-modify-write operation).
Otherwise, loads the actual value stored in *this into expected (performs load operation).

std::memory_order (Cont…)

Thread2 inserted a new node to the beginning of the list thus this is the new head of the list



Thread1 holds this pointer as the head of the list

MKBN

```
COMPARE_EXCHANGE_STRONG( T& old_value, T new_value,
                         MEMORY_ORDER on_success,
                         MEMORY_ORDER on_failure):
    T temp = atomic_value.load(on_failure);
    if(temp != old_value){
        old_value = temp;
        return false;
    }
    Lock l;
    temp = atomic_value.load(on_failure);
    if(temp != old_value){
        old_value = temp;
        return false;
    }
    atomic_value.store(new_value,on_success);
    return true;
}
```

```
#include <atomic>
template<typename T>
struct node{
    T data;
    node* next;
    node(const T& data) : data(data), next(nullptr) {}
};

template<typename T>
class stack{
    std::atomic<node<T>*> head;
 public:
    void push(const T& data)    {
      node<T>* new_node = new node<T>(data);

      new_node->next = head.load(std::memory_order_relaxed);

      while(!head.compare_exchange_weak(new_node->next, new_node,
                                        std::memory_order_release,
                                        std::memory_order_relaxed));
};
```

Default memory order

- If you don't specify any memory order, then the default one is the strongest order: `std::memory_order_seq_cst`.
  - It's expensive!
- In general: memory barriers are expensive!
- Note: not all platforms provide all barriers.

YouTube

CppCon 2017: Fedor Pikus
"C++ atomics, from basic to advanced. What do they really do?"

Thread safe Singleton

```
class Singleton {
public:
    static Singleton& instance(){
        if (!m_instance){
            m_instance = new Singleton;
        }
        return *m_instance;
    }
protected:
    Singleton();
private:
    static Singleton* m_instance;
};
```

Does it thread safe??

Thread safe Singleton (Cont…)

- What if **two threads** concurrently invoke the `instance()` method?

Thread One	Thread Two	Singleton
`Singleton.getInstance()`		`null`
	`Singleton.getInstance()`	`null`
`if (uniqueInstance == null)`		`null`
	`if (uniqueInstance == null)`	`null`
`uniqueInstance = new Singleton();`		`<object1>`
`return uniqueInstance;`		`<object1>`
	`uniqueInstance = new Singleton();`	`<object2>`
	`return uniqueInstance;`	`<object2>`

Thread safe Singleton (Cont…)

- Yup! Two instances of the singleton class could be created!

- Thread may be preempted by the operating system during `if (!m_instance).`
  Then another thread gets control.
  Since the first thread did not finished creation of the `m_instance`, the second thread evaluates `(!m_instance)` as `true` and creates another singleton.

Thread safe Singleton (Cont…)

- **How could we prevent this?**

- Several methods:
  - **Make the instance() synchronized**.
    - Synchronization is expensive, however, and is really only needed the first time the unique instance is created.

  - Do an **eager instantiation** of the instance rather than a lazy instantiation

Thread Safe Singleton – Solution #1 – Eager Initialization

- Here is Singleton with eager instantiation.

```
public class Singleton {
      private static Singleton uniqInstance = new Singleton();
      private int data = 0;

      public static Singleton instance() {
            return uniqInstance ;
      }
      private Singleton() {}
      // Accessors and mutators here!
}
```

Memory Consumer!!!

Thread Safe Singleton – Solution #2 - Locking

```
class Singleton {
public:
   static Singleton& instance(){
            m_key.Lock();
                  if (!m_instance)
                        m_instance = new Singleton;
            m_key.Unlock();
            return *m_instance;
   }
protected:
   Singleton();
private:
   static Singleton* m_instance;
   static std::mutex m_key;
};

//cpp
Singleton* Singleton::m_instance = 0;
std:mutex Singleton::m_key;
```

Inefficient!!!

```
class Singleton {
public:
   static Singleton& instance(){
                    if (!m_instance){
                           m_key.Lock();
                           m_instance = new Singleton;
                           m_key.Unlock();
                    }
             return *m_instance;
   }
protected:
   Singleton();
private:
   static Singleton* m_instance;
   static CCriticalSection m_key;
};

//cpp
Singleton* Singleton::m_instance = 0;
CCriticalSection Singleton::m_key;
```

Will it work?

Thread Safe Singleton – Solution #3 – Double Locking

```
class Singleton {
public:
    static Singleton& instance(){
                        if (!m_instance){
                                m_key.Lock();
                                if(!m_instance){
                                        m_instance = new Singleton;
                                }
                                m_key.Unlock();
                        }
                return *m_instance;
    }
protected:
    Singleton();
private:
    static Singleton* m_instance;
    static CCriticalSection m_key;
};
```

Now it is OK?

Solution3 – double checked locking (cont…)

- Nice idea? Now let's destroy the party…

"Double Checked Locking is Broken"

- The authors of the DCLP failed to consider a fundamental problem, that of *ensuring that the machine instructions executed during DCLP are executed in an acceptable order*.

Solution3 – double checked locking (cont…)

- Consider the line that initializes `pInstance`:

```
pInstance = new Singleton;
```

- This statement causes three things to happen:
  - Step 1: Allocate memory to hold a `Singleton` object.
  - Step 2: Construct a `Singleton` object in the allocated memory.
  - Step 3: Make `pInstance` point to the allocated memory.

Solution3 – double checked locking (cont…)

- But!!! Compilers are not constrained to perform these steps in this order!
  - In particular, compilers are sometimes allowed to swap step 2 and 3.

אין הבטחה
3-ו 2שפעולות
יתבצעו בסדר הזה,
אם בכלל.

```
void Foo() {
  int x = 0, y = 0;          // Statement 1
  x = 5;                     // Statement 2
  y = 10;                    // Statement 3
  printf("%d, %d", x, y);    // Statement 4
}
```

Solution3 – double checked locking (cont…)

- **The compiler might generate the following code:**

```
Singleton* Singleton::instance() {
  if (pInstance == 0) {
    Lock lock;
    if (pInstance == 0) {
      pInstance =                          // Step 3
        operator new(sizeof(Singleton)); // Step 1
      new (pInstance) Singleton;           // Step 2
    }
  }
  return pInstance;
}
```

Can cause accessing not constructed instance of Singleton!

אם Aנכנס ומבצע את 1 ו-3 ועוצר, ואז Bמגיע, הוא רואה שהמצביע לא null ולכן מחזיר אותו, אפילו שהוא עדין לא מאותחל.

Solving the DCLP with C++11

- One solution is to use memory barriers, making sure that the sequence of the instructions is not changed.

- Another solution is to declare the Singleton's pointer as **`atomic`**, thus we can let the other threads use it only after we release its value. otherwise, they will see the old value, which is `nullptr`.

Solution 1: Using C++11 Acquire and Release Fences

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_relaxed);
    std::atomic_thread_fence(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            std::atomic_thread_fence(std::memory_order_release);
            m_instance.store(tmp, std::memory_order_relaxed);
        }
    }
    return tmp;
}
```

Solution 2: Using C++11 Low-Level Ordering Constraints

```
std::atomic<Singleton*> Singleton::m_instance;
std::mutex Singleton::m_mutex;

Singleton* Singleton::getInstance() {
    Singleton* tmp = m_instance.load(std::memory_order_acquire);
    if (tmp == nullptr) {
        std::lock_guard<std::mutex> lock(m_mutex);
        tmp = m_instance.load(std::memory_order_relaxed);
        if (tmp == nullptr) {
            tmp = new Singleton;
            m_instance.store(tmp, std::memory_order_release);
        }
    }
    return tmp;
}
```

Hands-On Exercise: Define a non lock Queue

- Given a List, implement the function **`void push_front(T x)`** which inserts a new node with the value x to the beginning of the list.

```
template<class T>
struct Node{
    T _value;
    std::shared_ptr<Node>    _next;
};


template<class T>
class List
{
public:
    List() = default;
    ~List() = default;

    void push_front(T val){
        //TODO: implement
        //...
        //1. prepare the data
        //2. update the _head (do the swap of the value)
    }
private:
    std::atomic<std::shared_ptr<Node<T>>>  _head;
};
```

std::thread – Low Level Thread Handling

- std::thread is a class contains the API for using low-level threads, where the programmer take care of the creation, join/detach, mutex etc’.

Low level methods: std::thread

High level methods: Task

std::thread – creating a thread

- Has a constructor that gets a callable parameter:
  - a function pointer
  - a functor
  - a lambda function
- The first parameter is the callable object, and the others are parameters to that object.
- All parameters are passed-by-value by default but can also passed-by-reference.

std::thread – creating a thread (Cont…)

- Function Pointer:

```
void simpleFunction(){
    for(int i=0; i<10; ++i){
        std::cout<<i<<" ";
    }
    std::cout<<std::endl;
}

void test_simple_thread()
{
    //creating the thread with a function pointer:
    std::thread t1(simpleFunction);
    std::thread t2(simpleFunction);
    t1.join();
    t2.join();
}
```

- Function Object:

```
struct SimpleFunctor{
    void operator()(){
        for(int i=0; i<10; ++i){
            std::cout<<i<<" ";
        }
        std::cout<<std::endl;
    }
};

void test_simple_thread(){
    //creating the thread with a functor:
    SimpleFunctor func;
    std::thread t3(func);
    std::thread t4((SimpleFunctor()));
       //need to enclose the creation due to function
       //declaration conversion
    t3.join();
    t4.join();
}
```

- Lambda Function:

```
void test_simple_thread()
{
    auto lambdaFunc = [] {
        for(int i=0; i<10; ++i){
            std::cout<<i<<" ";
        }
        std::cout<<std::endl;
    };

    std::thread t5([] {
        for(int i=0; i<10; ++i){
            std::cout<<i<<" ";
        }
        std::cout<<std::endl;
    }
    );
    std::thread t6(lambdaFunc);
    t5.join();
    t6.join();
}
```

Waiting for threads to finish

- Once a thread has started, we may need to wait for the thread to finish before we can take some action.

- To wait for a thread, use the `std::thread::join()` function. This function makes the current thread wait until the thread identified by ***this** has finished executing.

- If we wish to separate the thread of execution from the thread object, allowing execution to continue independently, we can use `std::thread::detach()`.

Waiting for threads to finish (Cont…)

```
void independentThread(){
   std::cout << "starting concurrent thread...";
   std::this_thread::sleep_for(std::chrono::seconds(2));
   std::cout << "concurrent thread done!\n";
}

int main(){
   std::cout<<"---main thread---\n";
   std::cout << "creating a thread execution...\n";
   std::thread t(independentThread);
   t.detach();
   std::this_thread::sleep_for(std::chrono::seconds(1));
   std::cout << "--end main thread---\n";
   std::this_thread::sleep_for(std::chrono::seconds(5));
}
```

Specifying Arguments for the thread function

- The constructor of `std::thread` is a variadic function.
- The arguments are sent by value due to internal implementation. Sending by reference is available only using the reference wrappers `std::ref` or `std::cref`.

```
void func1(int& i){}  //error! will be OK with const&

int main(){
    int x = 3;
    std::thread t1(func1,x);
}
```

```
void func(MyClass m){
        std::cout<<"---inside func...\n";
        std::cout<<"---end of func...\n";
}

int main(){
    MyClass m;  //call default constructor
    std::cout<<"---create a thread with func...\n";
    std::thread theThread(func, m);  //parameters are passed by value to the
                                     //constructor, thus call copy constructor

    std::cout<<"---joining the thread...\n";
    if(theThread.joinable()){
        theThread.join();
    }
    std::cout<<"---end of main...\n";
}
```

```
inside default constructor
---create a thread with func...
inside copy constructor
inside move constructor
insdie ~MyClass()
inside move constructor
---joining the thread...
---inside func...
---end of func...
insdie ~MyClass()
insdie ~MyClass()
---end of main...
insdie ~MyClass()
```

```
void func(const MyClass& m){
        std::cout<<"---inside func...\n";
        std::cout<<"---end of func...\n";
}

int main(){
    MyClass m;  //call default constructor
    std::cout<<"---create a thread with func...\n";
    std::thread theThread(func, m);

    std::cout<<"---joining the thread...\n";
    if(theThread.joinable()){
        theThread.join();
    }
    std::cout<<"---end of main...\n";
}
```

```
inside default constructor
---create a thread with func...
inside copy constructor
inside move constructor
insdie ~MyClass()
---joining the thread...
---inside func...
---end of func...
insdie ~MyClass()
---end of main...
insdie ~MyClass()
```

same output as sending by value!

```
void func(MyClass& m){
        std::cout<<"---inside func...\n";
        std::cout<<"---end of func...\n";
}

int main(){
    MyClass m;  //call default constructor
    std::cout<<"---create a thread with func...\n";
    std::thread theThread(func, std::ref(m));

    std::cout<<"---joining the thread...\n";
    if(theThread.joinable()){
        theThread.join();
    }
    std::cout<<"---end of main...\n";
}
```

```
inside default constructor
---create a thread with func...
---joining the thread...
---inside func...
---end of func...
---end of main...
insdie ~MyClass()
```

Only one instance of MyClass was created!

Returning a value from the thread

- **Function ptr** – parameters are sent by value, but we can use `std::ref`/`std::cref` to send them by value.
- **Function objects** – we can send it by reference and put a variable inside the struct to hold the return value.
- **Lambda expression** – we can pass references to lambda functions using lambda capture.

std::mutex - Handling Race Condition

- `std::mutex` offers exclusive, non-recursive ownership semantics.
- A calling thread owns a mutex from the time that it successfully calls either `lock` or `try_lock` until it calls unlock.
- When a thread owns a mutex, all other threads will block (for calls to `lock`) or receive a false return value (for `try_lock`) if they attempt to claim ownership of the mutex.
  - A calling thread must not own the mutex prior to calling `lock` or `try_lock`.

```
std::mutex mtx;           // mutex for critical section

void print_block (int n, char c) {
  // critical section (exclusive access to std::cout signaled by locking mtx):
  mtx.lock();
  for (int i=0; i<n; ++i) {
      std::this_thread::sleep_for(std::chrono::milliseconds(50));
      std::cout << c;
  }
  std::cout << '\n';
  mtx.unlock();
}

int main (){
  std::thread th1 (print_block,50,'*');
  std::thread th2 (print_block,50,'$');

  th1.join();
  th2.join();
}
```

std::mutex - Handling Race Condition (Cont…)

- **Note**: The behavior of a program is **undefined** if a mutex is destroyed while still owned by any threads, or a thread terminates while owning a mutex.

- `std::mutex` is neither copyable nor movable.

std::lock_guard – Handling Deadlock

- A **lock guard** is an object that manages a mutex object by keeping it always locked.
- On construction, the mutex object is locked by the calling thread, and on destruction, the mutex is unlocked.
- Is especially useful as an object with automatic duration that lasts until the end of its context. In this way, it guarantees the mutex object is properly unlocked in case an exception is thrown.

```
std::mutex mtx;

void print_even (int x) {
  if (x%2==0) std::cout << x << " is even\n";
  else throw (std::logic_error("not even"));
}

void print_thread_id (int id) {
  try {
    std::lock_guard<std::mutex> lck (mtx);
    print_even(id);
  }
  catch (std::logic_error&) {
    std::cout << "[exception caught]\n";
  }
}
```

std::unique_lock wrapper

- A **unique lock** is an object that manages a mutex object with unique ownership in both states: locked and unlocked.

- When you want to lock a mutex, you create a local variable of type `std::unique_lock` passing the mutex as parameter.
- When the `unique_lock` is constructed, it will lock the mutex, and it gets destructed it will unlock the mutex.
- More importantly: If an exceptions is thrown, the `std::unique_lock` destructor will be called and so the mutex will be unlocked.

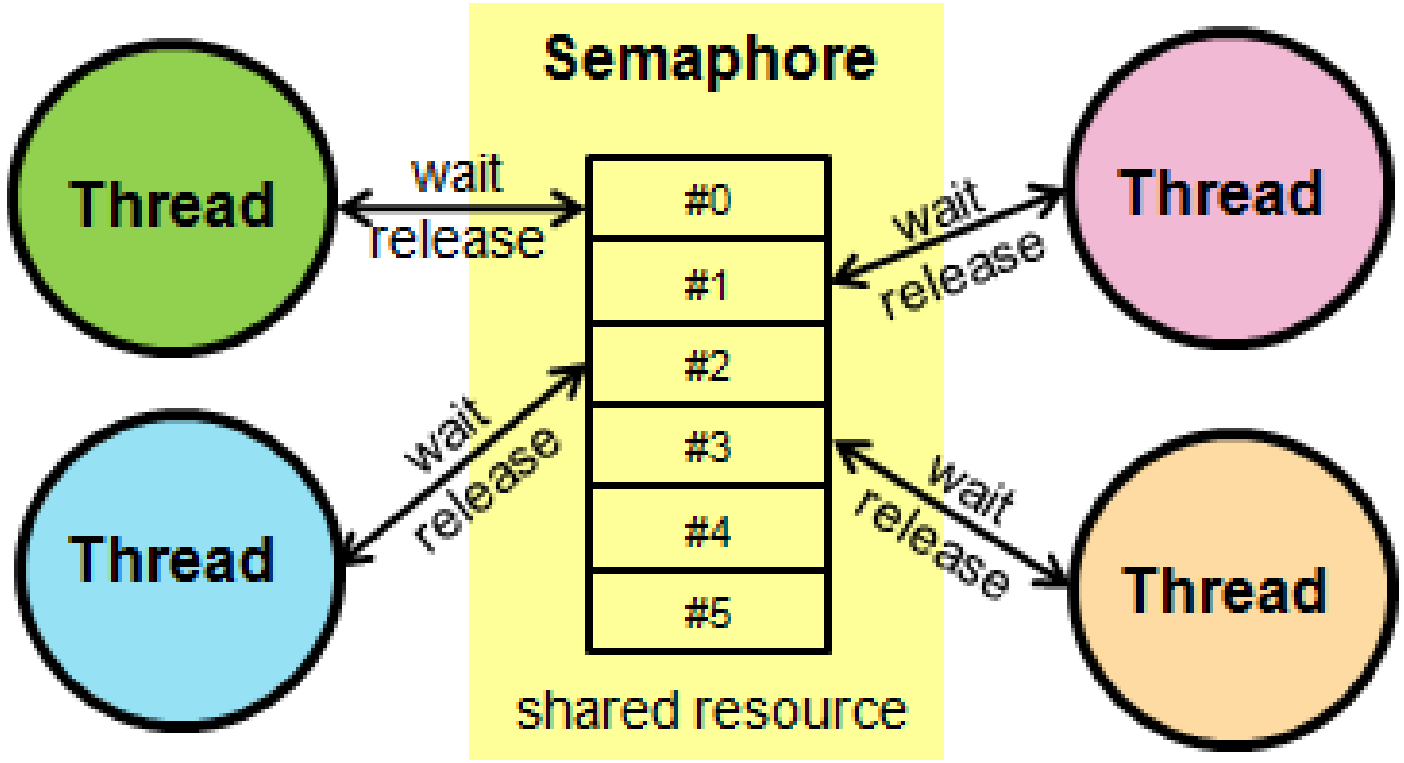
std::lock_guard vs. std::unique_lock

- **You can lock and unlock** a `std::unique_lock`. `std::lock_guard` will be locked only once on construction and unlocked on destruction.
- If you use conditional variable, you should use `std::unique_lock`.
- `std::unique_lock` **can be constructed without locking the mutex** immediately but to build the RAII wrapper.

```
std::unique_lock<std::mutex> lk1(mutex1, std::defer_lock);
std::unique_lock<std::mutex> lk2(mutex2, std::defer_lock);
std::lock(lk1, lk2);
```

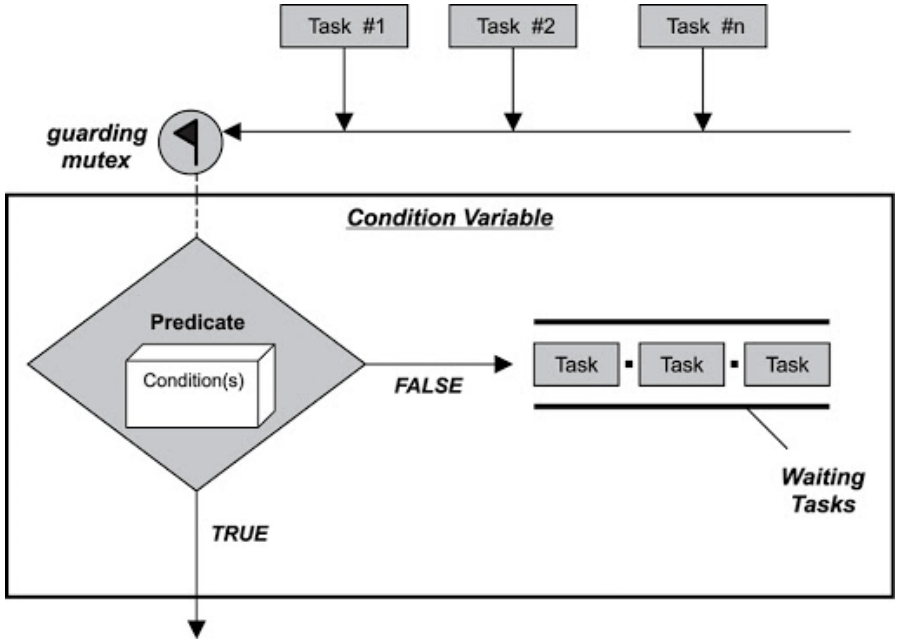
Semaphores

- Semaphore support was added only on **C++20**.
  - `std::counting_semaphore`, `std::binary_semaphore`



std::condition_variable

- The `std::condition_variable` class is a synchronization primitive that can be used to block a thread, or multiple threads at the same time, until another thread both modifies a shared variable (the condition) and notifies the `std::condition_variable`.



std::condition_variable (Cont…)

- **The thread that intends to modify the shared variable** has to
  - **acquire** a `std::mutex`.
  - **perform** the modification while the lock is held.
  - **notify** by executing `notify_one` or `notify_all` on the `std::condition_variable` (the lock does not need to be held for notification).

- Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.
- conditional variable usually use `std::unique_lock`.

std::condition_variable (Cont…)

- **`std::notify_one`** - If any threads are waiting on *this, calling notify_one unblocks one of the waiting threads.

- **`std::notify_all`** - Unblocks all threads currently waiting for *this.

std::condition_variable (Cont…)

- **Any thread that intends to wait on `std::condition_variable` has to:**
  - **acquire** a `std::unique_lock<std::mutex>`, on the same mutex as used to protect the shared variable.
  - either
    - **check the condition**, in case it was already updated and notified.
    - **execute** `wait`, `wait_for`, or `wait_until`. The wait operations atomically release the mutex and suspend the execution of the thread.
    - When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is **awakened**, and the mutex is atomically reacquired. The thread should then **check the condition** and resume waiting if the wake up was spurious.

std::condition_variable (Cont…)

- **`std::wait`** - causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied.

- Its parameter is the mutex shared by all threads, and a predicate (of type `bool pred()`) which returns false if the waiting should be continued.

```
std::condition_variable cv;
std::mutex cv_m;
int i = 0;

void waits(){
    std::unique_lock<std::mutex> lk(cv_m);
    cv.wait(lk, [index]{return i == 1;});
}

void signals(){
    {
        std::lock_guard<std::mutex> lk(cv_m);
    }
    cv.notify_all();
    {
        std::lock_guard<std::mutex> lk(cv_m);
        i = 1;
    }
    cv.notify_all();    //threads will wake up since now i==1
}
```

```cpp
std::mutex m;
std::condition_variable cv;
std::string data;
bool ready = false;
bool processed = false;

void worker_thread(){
    // Wait until main() sends data
    std::unique_lock<std::mutex> lk(m);
    cv.wait(lk, []{return ready;});

    // after the wait, we own the lock.
     data += " after processing";

    // Send data back to main()
    processed = true;

    lk.unlock();
    cv.notify_one();
}
```

```cpp
int main(){
    std::thread worker(worker_thread);

    data = "Example data";
    // send data to the worker thread
    {
        std::lock_guard<std::mutex> lk(m);
        ready = true;
    }
    cv.notify_one();

    // wait for the worker
    {
        std::unique_lock<std::mutex> lk(m);
        cv.wait(lk, []{return processed;});
    }

    worker.join();
}
```

std::native_handle()

- If you wish to involve a thread handling function from the underlying platform, you can ask for a handler to it.
  - For example: to enable realtime scheduling of C++ threads.

- Not all compilers implements this. It is optional.

Hands-On Exercise – Implement a Semaphore

- Implement a Semaphore using methods from the library std::thread.

- Reminder: Your semaphore should support:
  - **Constructor** – with a counter representing the number of threads can be active at a time.
  - **wait** – puts on a sleep the thread holding the mutex if the number of active threads is already equal to count.
  - **signal** – called by a thread when he finishes handling the critical section.

High Level Concurrency (Task Based Concurrency)

- **Task** – a function that is executed automatically in a different thread, without the need to create the thread.

- Tasks consist of two components, which are connected by a communication channel:
  - One component as endpoint of the channel produces the result.
  - The other endpoint consumes it.
  - The producer is called **Promise**, the consumer **Future**.

High Level Concurrency (Task Based Concurrency) (Cont…)

- `std::future` provides access o the result of the callable.

- `std::async` – the mechanism for running a task. Can get a function object, function pointer or lambda expression.
  It returns an object of type `std::future`.

- Include file : `<future>`

Creating a task

```
void printNumbers(){
    for(int i=0; i<10; i++){
        std::this_thread::sleep_for(200ms);
        std::cout<<i<<"..";
    }
    std::cout<<std::endl;
}

int main(){
    auto res = std::async(printNumbers);
    std::cout<<"--main finished--\n";
}
```

High Level Concurrency (Task Based Concurrency) (Cont…)

- With the `std::async` library, manual thread handling is abstracted away, and you rely on the library's system to possibly spawn threads, depending on available resources.

- The main benefit of this form of parallelism is the great ease in getting returned values from tasks that you start.

- **Passing data to thread:**
  By default, arguments are sent to `std::async` by value.
  Use `std::ref` or `std::cref` to send them by reference.

- **To get value from a task:**
  simply return it from the corresponding function.
  The value is available through the `std::future` object.

std::async

- The function template **`std::async(func)`** runs the function or a function object `func` asynchronously (potentially in a separate thread which might be a part of a thread pool) and returns a `std::future` that will eventually hold the result of that function call.

- Arguments expected by function can be passed to **`std::async()`** as arguments after the function pointer argument.

std::async (Cont…)

- We can pass any callback in std::async i.e.
  - Function Pointer
  - Function Object
  - Lambda Function

- There are two constructors to std::async:

```
std::future<return_type> async(Callable, Args);

std::future<return_type> async(Launch_Policy, Callable, Args);
```

std::future

- `std::future` is the objects that is **returned** by asynchronous operations.
- Used to communicate between threads, together with a std::promise object.

- Syntax: `std::future<returned_type_of_thread>`

- A class template that stores a value that will be assigned in the future and provides a way to access that value (with `get()`).

std::future (Cont…)

```
int factorial(int num){
    int result = 1;
    for(int i=2; i<=num; i++){
        result*=i;
    }
    return result;
}

int main(){
    std::future<int> res = std::async(factorial,10);

    if(res.valid()){
        std::cout<<"getting value again?\n";
        value = res.get();
    }
}
```

std::future (Cont…)

- If its value is accessed before the value is assigned, it will block until the value resolves. It will make the main thread wait for the thread.

- The task is assigning the return value inside a `std::promise` object, and this data is available through a `std::future` object in the caller thread.

- promise/future pair allow safe data sharing between threads without requiring any other explicit synchronization.

std::future (Cont…)

- Each future's `get()` method can only be called once.

- If you want a future that can be accessed multiple times, use a `std::shared_future` instead.
- Otherwise, initialize a different promise future pair.

- If you don’t assign the returned future object from the `std::async` execution, the main thread will be blocked until the thread will finish.

std::future (Cont…)

- We can use `std::wait()` on the `std::future` object just to block the main thread without getting the shared data.
  - This block will be release when the future object is assigned with the return value.
- We can use `std::wait_for` in order not to block the main thread until std::future is ready.
- Similarly we can use `std::wait_until` and sent as parameter a time point starting from a certain time stamp.

std::future (Cont…)

```
int main(){
    std::cout<<"--main started--\n";

    std::future<int> res = std::async(factorial,10);

    std::this_thread::sleep_for(1s);
    std::cout<<"--main is waiting for the task--\n";

    if(res.valid()){
        res.wait();
        auto value = res.get();
        std::cout<<"return value is: "<<value<<std::endl;
    }

    std::cout<<"--main finished--\n";
}
```

std::future (Cont…)

```
void test2_wait_for_with_loop()
{
    std::future<int> res = std::async(std::launch::async,sumUntil,15);

    if(res.valid()){

        auto status = res.wait_for(1s); //try with 5s
        while(status == std::future_status::timeout){
            status = res.wait_for(1s);
        }
        auto value = res.get();
        std::cout<<"value is ready: "<<value<<std::endl;
    }
}
```

std::future (Cont…)

```
void test_wait_until_with_loop(){

  std::future<int> res = std::async(std::launch::async,sumUntil,15);
  if(res.valid()){
      auto timePoint = std::chrono::system_clock::now();

      std::future_status status;

      do{
         timePoint += 1s;
         status = res.wait_until(timePoint);
      }while(status == std::future_status::timeout);

      auto value = res.get();
      std::cout<<"value is ready: "<<value<<std::endl;
   }
}
```

std::promise

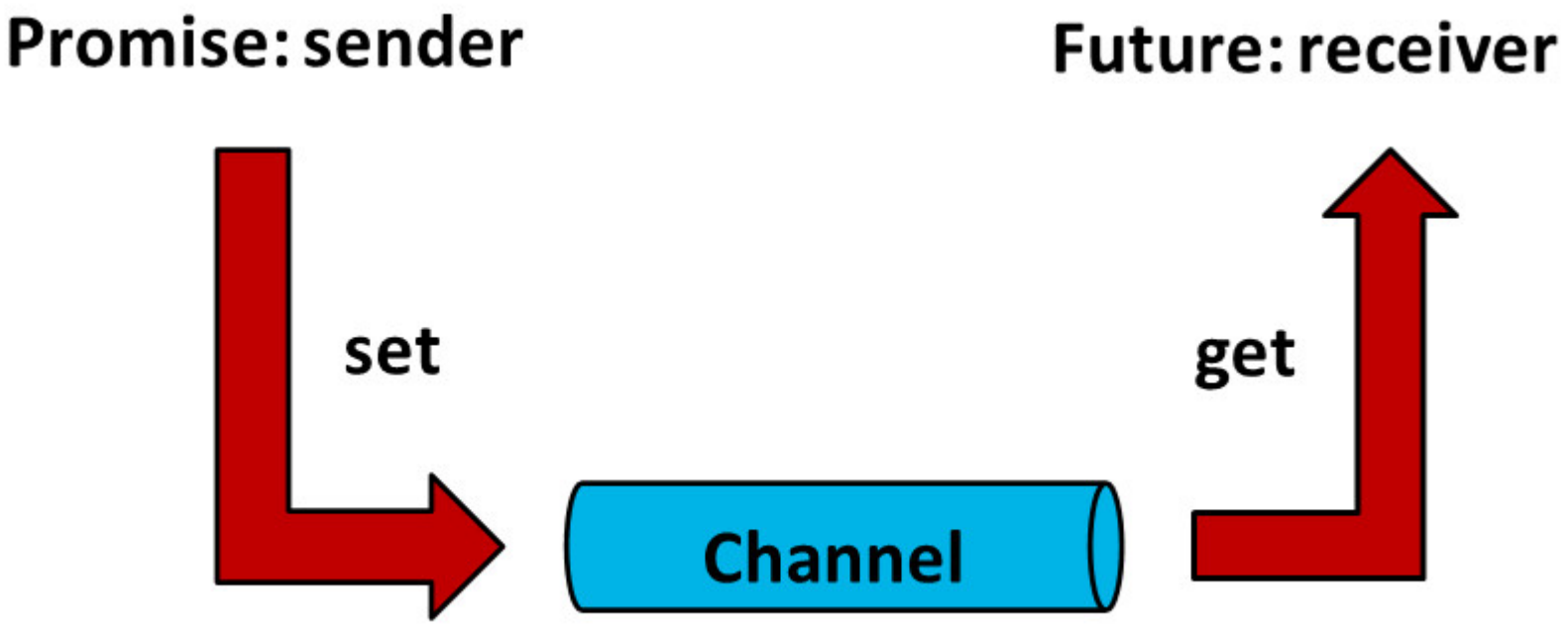
- Provides a facility to store a value or an exception that is later acquired asynchronously via the future **that the promise creates**.

- Syntax: `std::promise<data_type>`
- `std::set_value` – sets a value to the promise object and release the block of the thread that tries to get its value. Can be done only once!

- Every promise **is associated with a future**! And a promise **sets** the value of that future.

std::promise (Cont…)

- If your promise object is destroyed before you set its value, the `get()` method for its associated future will throw an exception.



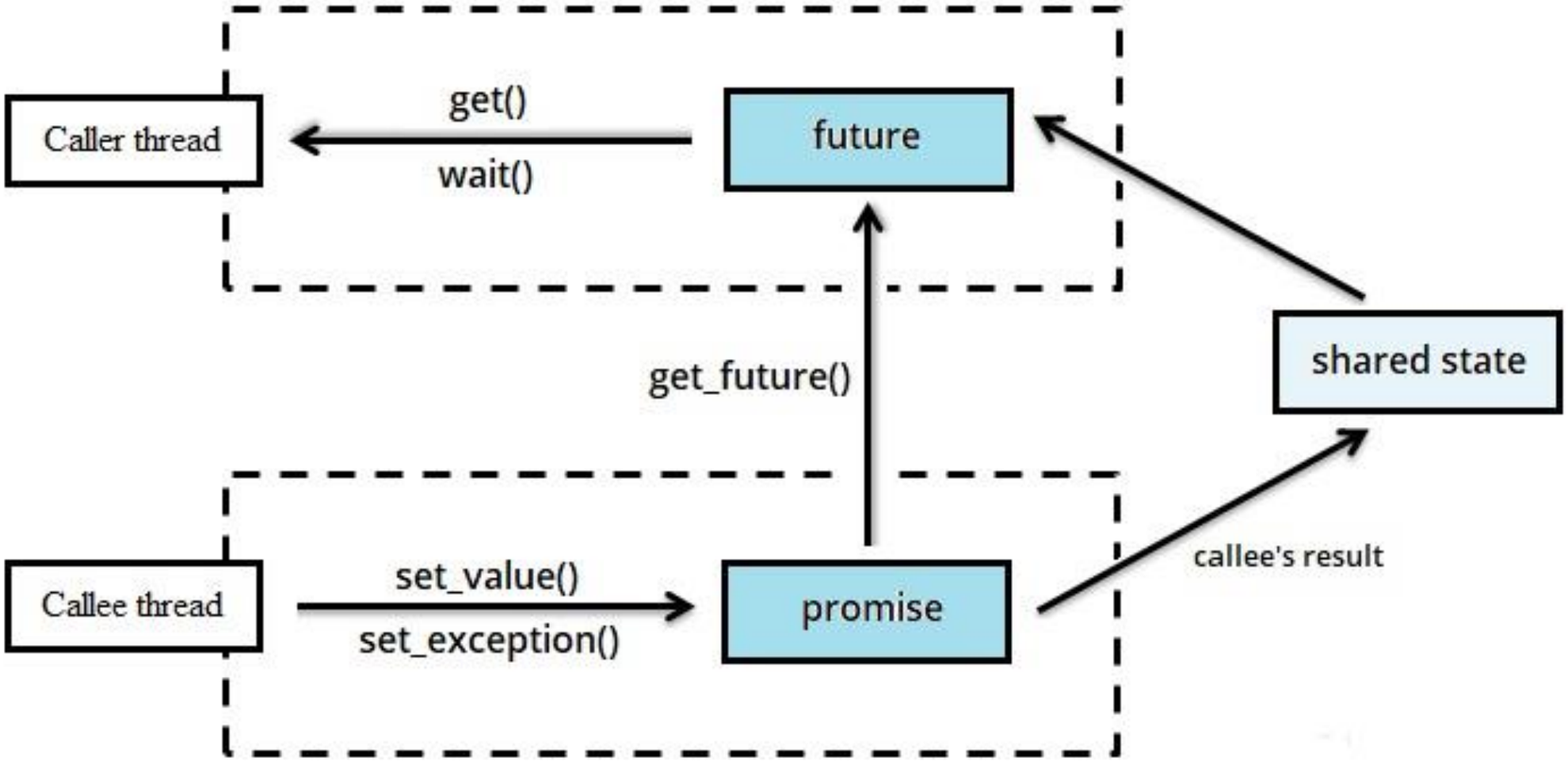
```
int sumUntil(std::promise<int>& num){
    auto val = num.get_future();
    auto count = val.get();
    int result = 0;
    for(int i=1; i<=count; i++){
         result+=i;
    }
    return result;
}

int main(){
    std::promise<int> num;
    std::future<int> res = std::async(std::launch::async, sumUntil, std::ref(num));
    //num is not ready test since we didn't call set_value yet
    //thus the task is blocked and wait

    num.set_value(10);

    if(res.valid()){
        auto sum = res.get();
    }
}
```

High Level Concurrency (Task Based Concurrency) (Cont…)



Launch Policies

- The first argument to `std::async` is the **launch policy**, which controls the asynchronous behavior of `std::async`.

- **std::launch::async -** passed function will be executed in separate thread.
- **std::launch::deferred –** passed function will be called when other thread will call `get()` on `future` to access the shared state.
- **std::launch::async | std::launch::deferred** - the default behavior. With this launch policy it can run asynchronously or not depending on the load on system. But we have no control over it.

Launch Policies (Cont...)

```
void printNumbers(){
    for(int i=0; i<10; i++){
        std::this_thread::sleep_for(200ms);
        std::cout<<i<<"..";
    }
    std::cout<<std::endl;
}

int main(){
    // auto res = std::async(std::launch::async, printNumbers);
    auto res = std::async(std::launch::deferred, printNumbers);
    std::this_thread::sleep_for(1s);
    std::cout<<"--main is waiting for the task--\n";
    res.get();
    std::cout<<"--main finished--\n";
}
```

Propagating exceptions across threads

- Using `std::future` enables us to catch an exception in a worker thread and re-throw it in the main thread that's waiting for the worker to finish.

- When exception it handled (typically, in a catch clause), the function `std::current_exception()` can be called to capture the current exception object into an `std::exception_ptr` that holds either a copy or a reference to that exception object (depending on the implementation).
  - The referenced object remains valid at least as long as there is an exception_ptr object that refers to it.

```
//source: https://vorbrodt.blog/2019/03/24/propagate-exceptions-across-threads/
int main(){
    std::exception_ptr ep = nullptr;
    std::thread t([&](){
        try{
            std::stringstream str;
            str << std::this_thread::get_id();
            throw std::runtime_error(str.str().c_str());
        }
        catch(...){
             ep = std::current_exception();
        }});
    t.join();
    //check the exception caught in the main thread
    if(ep != nullptr){
        try{
            rethrow_exception(ep);
        }
        catch(std::exception& e)        {
            std::cout << "[Thread " << std::this_thread::get_id() <<
                        "]: caught exception from thread " << e.what() << std::endl;
        }
    }
}
```

std::async vs. std::thread

- Using `std::async` is a convenient way to fire off a thread for some asynchronous computation and marshal the result back via a future. `std::thread` does not provide an easy way to return a value from a thread.

- If you want a way to detect (via polling) whether an asynchronous job is done:
  with `std::thread` you must manage it yourself.
  with `std::async` you can query `std::future::valid()` (or use `std::future::wait_for`/`wait_until(...)`) to know when it is done.

std::function

- Class template `std::function` is a general polymorphic function wrapper.
  - The instance of `std::function` can store, copy and call any callable target -- function, lambda expression, bind expression or other function object, as well as pointer to member function and data member.

- Syntax: `std::function<return_value(args_type)> func;`
- Header file: `<functional>`

- `std::bind(func,args)` - Returns a function object based on *func*, but with its arguments bound to *args*.

std::function (Cont…)

```
//regular function
void print_num(int i) { std::cout << i << '\n'; }
void some_func(int i, double d) {}

int main()
{
    // Store free function
    std::cout << "---Store free function\n";
    std::function<void(int)> f_display = print_num;
    f_display(-9);

    std::function<void(int,double)> f_someFunc = some_func;
}
```

std::function (Cont…)

- `std::function` can also store a lambda expression:

```
int main()
{
   // Storage lambda
   std::cout << "---Store lambda\n";
   std::function<void()> f_display_42 = []() { print_num(42); };
   f_display_42();

   std::function<void(int)> f_display_lambda =
                [](int i) { std::cout<<i<<std::endl; };
   f_display_lambda(10);
}
```

std::function (Cont…)

- And can also store a member function. In this case we need to specify the `this` argument:

```
struct Foo {
    Foo(int num) : _num(num) {}
    void print_add(int i) const { std::cout << _num+i << '\n'; }
    int _num = 1;
};

int main(){
    // Calls stored in member functions
    std::function<void(const Foo&, int)> f_add_display = &Foo::print_add;
    const Foo foo(2000);
    f_add_display(foo, 1);
    f_add_display(1000, 1); // A foo object is constructed!
}
```

std::bind

- The template function `std::bind(fn,args)` returns a function object based on `fn`, but with its arguments bound to `args`.

```
double my_divide (double x, double y) {return x/y;}

int main () {
  using namespace std::placeholders;    // adds visibility of _1, _2, _3,...

  // binding functions:
  auto fn_five = std::bind (my_divide,10,2);      // returns 10/2
  std::cout << fn_five() << '\n';                 // 5

  auto fn_half = std::bind (my_divide,_1,2);      // returns x/2
  std::cout << fn_half(10) << '\n';               // 5

  auto fn_invert = std::bind (my_divide,_2,_1);   // returns y/x
  std::cout << fn_invert(10,2) << '\n';           // 0.2
}
```

std::bind (Cont…)

- The `std::placeholders` namespace contains the placeholder objects [_1, . . . _N] where N is an implementation defined maximum number.

- When used as an argument in a `std::bind` expression, the placeholder objects are stored in the generated function object, and when that function object is invoked with unbound arguments, each placeholder `_N` is replaced by the corresponding Nth unbound argument.

std::packaged_task

- A `packaged_task` wraps a callable element and allows its result to be retrieved asynchronously.
- It is similar to `std::function` but transferring its result automatically to a future object.
- A `packaged_task` won't start on it's own, you have to invoke it:

```
std::packaged_task<int()> task(func);

auto f = task.get_future();
task(); // invoke the function
```

std::package_task (Cont…)

- A `packaged_task` needs to be invoked before you call the future's `get()`, otherwise you program will freeze as the future will never become ready:

```
std::packaged_task<int(int,int)> task(...);

auto f = task.get_future();

std::cout << f.get() << "\n"; // oops!

task(2,3);
```

std::package_task vs. std::async

- Use `std::async` if you want some things done and don't really care when they're done, and `std::packaged_task` if you want to wrap up things in order to move them to other threads or call them later.
  - A `std::packaged_task` is just a low-level feature for implementing `std::async` (which is why it can do more than `std::async` if used together with other low level stuff, like `std::thread`).
  - A `std::packaged_task` is a `std::function` linked to a `std::future` and `std::async` wraps and calls a `std::packaged_task` (possibly in a different thread).

std::package_task vs. std::async (Cont…)

- By using `std::async` you cannot run your task on a specific thread anymore, where `std::packaged_task` can be moved to other threads.

```
std::packaged_task<int(int,int)> task(...);
auto f = task.get_future();
std::thread newThread(std::move(task),2,3);

std::cout << f.get() << "\n";
```

Task based sync - Hands-On Exercise

- Write a program that will run 2 tasks:
  - Task 1 will create a std::vector<int> , fill it with the numbers 1,…,25 and return it.
  - Task 2 will get that vector and sum its elements.

- Task 2 should start running immediately as soon as the vector from Task 1 is ready.

Implementing a Thread Pool

```
class ThreadPool
{
public:
    //1. constructor that gets the pool's size and creates the threads

    //2. addTask method

private:
    //1. vector of threads

    //2. queue of tasks

    //3. some synchronization stuff...
};
```

std::optional

std::optional

- **Until C++17:** If a function may get a value or not, usually we use default values. But can we know inside the function if the value is valid (sent) or not (defaulted)?
  - We can add a Boolean parameter that will indicate that…

```
void maybe_take_an_int(int value = -1, bool is_valid = false);
void or_even_better(pair<int,bool> param = std::make_pair(-1, false));
pair<int, bool> maybe_return_an_int();
```

?

- What if we do not want to return any value in some cases (like fail)? We force a Boolean, an invalid object (may be expensive) etc'.

std::optional (Cont…)

- **C++17 solution**: `std::optional<T>` object that manages an optional contained value.
- Any instance of `std::optional<T>` at any given point in time either contains a value or does not contain a value.
- If an `std::optional<T>` contains a value, the value is guaranteed to be allocated as part of the optional object footprint.
  - no dynamic memory allocation ever takes place. Thus, an optional object models an object, not a pointer, even though `operator*()` and `operator->()` are defined.

std::optional (Cont…)

- When an object of type optional<T> is contextually converted to bool, the conversion returns true if the object contains a value and false if it does not contain a value.

- header file: **`<optional>`**

- Syntax: `std::optional<type>`

std::optional (Cont…)

- Initializing an optional object:

```
int main()
{
    //--- initialize an optional ---//
    std::optional<int> op1; //empty one
    std::optional op2{4};   //direct initialization using CTAD
    std::optional op3 = std::make_optional(5);

}
```

std::optional (Cont…)

- Can be used as the return type of a function or as a parameter to a function.
- Syntax: either return a value, or return `{ }` or `std::nullopt`.

- Storing references inside the optional is not allowed.

std::optional (Cont…)

```
std::optional<std::string> readFile(const std::string& fName)
{
    std::ifstream fin(fName);
    if(!fin.is_open()){
        //logic error: no such file or directory.
        //what if we do not want to use exceptions?
        return{};
    }

    std::string content;
    //read the file into content string...
    return content;
}
```

std::optional (Cont…)

- In order to check if there is a return value of the function:

```
//--- check if there is a return value ---//
 auto res = readFile("not_a_real_file");

//we can use the has_value method
 if(!res.has_value()){
     std::cout<<"something went wrong with this file...\n";
 }
 //we can use the bool operator
 if(!res){
     std::cout<<"something went wrong with this file...\n";
 }
```

std::optional (Cont…)

- In order to extract the return value:

```
if(auto num = getNumber()){
    //use the value() method
    std::cout<<"number is: "<<num.value()<<std::endl;
    //use operator*
    std::cout<<"number is: "<<*num<<std::endl;

    // std::cout<<"number is: "<<num<<std::endl;   //error!
}
```

std::optional (Cont…)

- If we try to extract the data but the `std::optional` object is empty, an exception will be thrown:

```
try{
    auto res = getNumber();
    //next line throws an exception!
    std::cout<<"trying to print the empty optional: "
            <<res.value()<<std::endl;
    //but this one is undefined behavior:
    std::cout<<*res;
}catch(std::bad_optional_access& e){
    std::cout<<"[exception] "<<e.what()<<std::endl;
}
```

Class Template Argument Deduction (CTAD)

Class Template Argument Deduction (CTAD)

- **Until C++14**: when calling a template function, the compiler can deduce the type T out of the parameter we sent. But we cannot do it with classes.

```
//until c++17 the compiler cannot deduce the int
//out of the initializer list.
std::vector<int> v{1,2,3,4};
```

Class Template Argument Deduction (CTAD)

- **Since C++17**: code verbosity is reduced. It extends template argument deduction to the construction of an object given only the name of a class template.

```
//since C++17:
std::vector v2{1,2,3,4};
std::tuple p{1,2};
std::mutex m;
std::lock_guard lk{m}; //instead ofstd::lock_guard<std::mutex>
```

Class Template Argument Deduction (CTAD)

- That works also for classes:

```
template<typename T>
class MyClass
{
public:
    MyClass(const T&){}
};

MyClass m1{5};  //instead of MyClass<int> m1{5};
auto pm = new MyClass{4.5};    //instead of new Data<double>{4.5};
```

C++17 Deduction Guides

- Template deduction guides are **patterns** associated with a template class that tell the compiler how to translate a set of parameter (and their types) into template arguments.

- After the class's declaration we can add some deduction guides.
- Syntax:

```
ClassName(given_type) -> ClassName<wanted_type>
```

```
template<typename T>
class MyClass{
public:
    MyClass(const T& val) : _val{val}{}
    T _val;
};
MyClass(const char*) -> MyClass<std::string>;
MyClass(int) -> MyClass<long>;

int main(){
    MyClass m{"hello"};
    MyClass k{5};

    std::cout<<typeid(m._val).name()<<std::endl;
    std::cout<<typeid(k._val).name()<<std::endl;
}
```

C++17 New Order of Evaluation Rules

C++17 New Order of Evaluation Rules

- Until C++17: the evaluation order of sub-expressions in function parameters was unpredictable.

```
std::cout << a() << b() << c();

func(std::unique_ptr<B>{new A}, std::unique_ptr<B>{new B}){}
```

C++17 New Order of Evaluation Rules (Cont…)

```
func(std::unique_ptr<B>(new A), std::unique_ptr<B>(new B)){}
```

- There are four operations that happen here during the function call
  - (1) new A
  - (2) unique_ptr<A> constructor
  - (3) new B
  - (4) unique_ptr<B> constructor
- The ordering of these was completely unspecified.
- **In C++17**, the new rules prohibit interleaving.
  - This leaves us with two valid orderings: (1), (2), (3), (4) or (3), (4), (1), (2).

Feature Test Macro (C++17)

- `__has_include` – used to check if a header file is available for inclusion or not. It helps track the progress of partial implementation of new C++ standards.

```
#ifdef __has_include
#   if __has_include(<filesystem>)
#       include <filesystem>
        namespace fs = std::filesystem;
#   else
#       include <experimental/filesystem>
        namespace fs = std::experimental::filesystem;

#   endif
#endif
```

if & switch with initialization

if & switch with initialization

- **Until C++17:** the scope of the index variable of a for loop is only the for's body. Like that we do not polluting the outer scope with unused variables. **What about if/switch statement?**

- **Since C++17:** we can use a similar semantic as in for loop inside the if statement.

- **Syntax:** `if(var_declaration_and_init ; the_condition){…}`

if & switch with initialization (Cont…)

```
void test(const std::string& message)
{
    if(std::ofstream fout("output.txt"); !message.empty()){
        fout<<message;
    }
    else{
        fout<<"empty message";
    }
    //here the fout is not recognized and destroyed (file closed)
}
```

if & switch with initialization (Cont…)

- The variable defined in the if statement is also visible in the else block.

```
void test4(){
    const std::string myString = "My Hello World Wow";

    //we need the variable it only for the if
    if (const auto it = myString.find("Program"); it != std::string::npos){
        std::cout << it << " Program\n";
    }
    else{
        std::cout << "Try another string since we got index "<<it<<"...\n";
    }
    //it is not declared in this scope
}
```

if & switch with initialization (Cont…)

```
void test5()
{
  using mySet = std::set<std::pair<std::string,int>>;
  mySet set1;
  std::pair<std::string,int> itemsToAdd[3]{{"hello",1},{"world",1},{"world",2}};
  for(auto &p : itemsToAdd)
  {
     // if-initializer + structured binding
     if(const auto [iter,inserted] = set1.insert(p);inserted)
     {
        std::cout << "Value(" << iter->first << ", " << iter->second << ") was inserted\n";
     }
     else
     {
        std::cout << "Value(" << iter->first << ", " << iter->second << ") was not inserted\n";
     }
  }
}
```

```
struct Device{
    enum class State_e{SLEEP, READY, BAD};
    State_e state() const {return _state;}

    operator State_e() {return _state;} //do we need this?
private:
    State_e _state = State_e::READY;
};
Device get_device(){ return Device(); }

int main(){
    //In order to do this we need neither:
    //1. declare dev in the outer scope and here test dev.state()
    //2. or supply a conversion from the device to enum type
    switch (Device dev = get_device())     {
       case Device::State_e::SLEEP: /*...*/ break;
       case Device::State_e::READY: /*...*/ break;
       case Device::State_e::BAD: /*...*/ break;
    }
}
```

```
struct Device{
    enum class State_e{SLEEP, READY, BAD};
    State_e state() const {return _state;}
private:
    State_e _state = State_e::READY;
};
Device get_device(){ return Device(); }

int main(){
    // the C++17 init-
statement syntax can be helpful when there is
    // no implicit conversion to integral or enumeration type
    switch (Device dev = get_device(); dev.state())    {
        case Device::State_e::SLEEP: /*...*/ break;
        case Device::State_e::READY: /*...*/ break;
        case Device::State_e::BAD: /*...*/ break;
    }
}
```

inline variables

inline variables

- **Before C++17**: when declaring a static member variable, we need to instantiate it in a cpp file, and sometimes we create a cpp file only for this reason.

- **C++11** and `constexpr` keyword allow you to declare and define static variables in one place, but it's limited to constexpr'essions only.

inline variables (Cont…)

- Previously only methods/functions could be specified as inline, but now you can do the same with variables, inside a header file:
  - "A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is used, and the behavior of the program is as if there was exactly one variable."

```
struct S{
    //we can initialize inside the class only literal types!
    const static std::string _str = "Hello World";  //ERROR!
};
```

inline variables (Cont…)

```
//the right way to do this:
struct S{
    const static std::string _str;
};
//should be in a cpp file:
const std::string S::_str = "Hello World";
```

```
//in c++17 we can do it better:
struct S{
    //we can initialize inside the class only literal types!
    inline const static std::string _str = "Hello World";
};
```

inline variables (Cont…)

- Note that `constexpr` variables are inline implicitly, so there's no need to use `constexpr inline myVar = 10;`.
- With this feature, a lot of header only libraries can limit the number of hacks (like using inline functions or templates) and just use inline variables.
- The advantage of inline varibels over `constexpr` is that your initialization expression doesn't have to be `constexpr`.

File System

File System

- Finally on **C++17** there is a standard file system library. Modeled directly over the boost file system.

- The library is located in the **`<filesystem>`** header. It uses namespace `std::filesystem`.
- Depending on the version of your compiler you might need to use `std::experimental::filesystem` namespace.

filesystem (Cont…)

- We have four core parts:
  - **The `path` object** – enables manipulating filesystems paths.
  - `directory_entry` – stores additional data as count, status, file size, last write time etc'.
  - `directory_iterators` – iterates over the directory_entry (not recursive).
    - `recursive_directory_iterator` – also visits subdirectories.
  - `file_status` – represents file type and permissions.
- **Does not replace the stream library class for creating, reading and writing files**.

iterating over all elements of a given path

- We can extract from a given path all its elements:

```
void test3()
{
    fs::path pathToShow(R"(C:\Windows\system.ini)");
    int i = 0;
    for (const auto& part : pathToShow){
        std::cout << "path part: " << i++ << " = " << part << "\n";
    }
}
```

```
path part: 0 = "C:"
path part: 1 = "\\"
path part: 2 = "Windows"
path part: 3 = "system.ini"
```

Composing a path

- By using methods:
  - **append** - adds a path with a directory separator.
  - **concat** - only adds the ‘string’ without any separator.

- By using operators: operator /=, or operator +=.

```
void test(){
        fs::path root {"/"};
        fs::path dir {"var/www/"};
        fs::path index {"index.html"};

        fs::path pathToIndex1 = root / dir / index;

        fs::path pathToIndex2 = root;
        pathToIndex2+= "var/www/";
        pathToIndex2+= "index.html";

        fs::path pathToIndex3 = root;
        pathToIndex3.append("var/www/").append("index.html");

        std::cout << pathToIndex1 << std::endl;
        std::cout << pathToIndex2 << std::endl;
        std::cout << pathToIndex3 << std::endl;
}
```

directory_entry – holds extra information about a directory

```
int main()
{
    fs::directory_entry entry{ R"(C:\temp\folder1\file1.txt)" };

    std::cout<<"is directory? "<< std::boolalpha <<
                entry.is_directory() << std::endl;
    std::cout<<"file size: "<< entry.file_size() << "bytes" <<
                std::endl;

    std::cout << entry << '\n';
    entry.replace_filename("hello.txt");
     //This function does not commit any changes to the filesystem!
    std::cout << entry << '\n';
}
```

filesystem (Cont…)

- Supports `directory_iterator` to iterates files in a directory. The iterator returns a `directory_entry` object.

```
void test2_directory_iterator(const std::string& filePath)
{
  fs::path currPath{filePath};
  std::cout<<"The content of the directory "<<currPath.string()
           <<":\n";
  for(const auto& dir_entry : fs::directory_iterator{currPath}){
      std::cout<< dir_entry.path().string() <<std::endl;
  }
}
```

filesystem (Cont…)

- We can check the type of the directory element:

```
void test3_directory_iterator_with_data(const std::string& filePath)
{
    fs::path currPath{ filePath };
    std::cout<<"The content of the director "<<currPath.string()<<": \n";
    for(const auto& dir_entry : fs::directory_iterator{currPath}){
        // std::cout<< dir_entry.path().string() <<std::endl;
        switch(const auto& f = dir_entry.path(); dir_entry.status().type()){
            case fs::file_type::directory:
                std::cout<<"[DIR]\t"<<f.string()<<std::endl;
                break;
            case fs::file_type::regular:
                std::cout<<'\t'<<f.string()<<'\t'<<dir_entry.file_size()<<std::endl;
                break;
        }
    }
}
```

filesystem error handling

- Most of the libraries throws an exception of type `filesyste::filesystem_error`.
- There are a few methods that returns an error value of type `std::error_code`.

filesystem::filesystem_error

```
int main()
{
    std::string filePath{ R"(C:\temp)" };
    try{
        test1_print_file_size(filePath);
    }catch(fs::filesystem_error& e){
       std::cout<<"[Exception]: "<<e.what()<<std::endl;
    }
    try{
        test1_print_file_size("dont_exists_file");
    }catch(fs::filesystem_error& e){
        std::cout<<"[Exception]: "<<e.what()<<std::endl;
    }
}
```

using std::error_code

- `std::error_code` was added on **C++11**.
- This object holds an error code originating from the OS or some low-level interface. It has a bool operator thus can be used in an if statement.
- It is platform dependent.

```
int main(){
    std::error_code ec{};
    fs::file_size("dont_exists_file",ec);
    if(ec){
        std::cout<<"Error! "<<ec.message()<<std::endl;
    }
}
```

filesystem::permissions - Handling File Permissions

- Some platforms enables manipulating the file's permissions.

```
void demo_perms(fs::perms p){
    std::cout << ((p & fs::perms::owner_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::owner_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::owner_exec) != fs::perms::none ? "x" : "-")
              << ((p & fs::perms::group_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::group_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::group_exec) != fs::perms::none ? "x" : "-")
              << ((p & fs::perms::others_read) != fs::perms::none ? "r" : "-")
              << ((p & fs::perms::others_write) != fs::perms::none ? "w" : "-")
              << ((p & fs::perms::others_exec) != fs::perms::none ? "x" : "-")
              << '\n';
}
int main(){
    fs::path file{"files/test.txt"};

    fs::permissions(file, fs::perms::owner_all, fs::perm_options::remove);
    auto perm = fs::status("test.txt").permissions();
    demo_perms(perm);
}
```

FileSystem - Hands-On Exercise

- Write a code that opens a given path and changes all folder names to FolderX, where X is an index (0,1,…), and all file names to FileY, where Y is another index (0,1,…).

- Print the folder's content before and after.

Modern C++ Alignment

Modern C++ Alignment

Before modern C++ the only way to align variables or structures on a given byte boundary was to inject padding;

To align a struct to 16 bytes you had to do this:

```
struct Old
{
    int x;
    char padding[16 - sizeof(int)];
};
```

Modern C++ introduced a keyword just for that: `alignas`.

Now you can specify struct's alignment like this:

```
struct alignas(16) New
{
    int x;
};
```

You can also space local variables apart, as well as struct/class members:

```
alignas(8)   int a;
alignas(256) long b[4];
```

Modern C++ Alignment (Cont…)

We can use `alignof` to find out the alignment of a type in the current platform:

```
// Possible output is 4
std::cout << alignof(int) << '\n';
```

Modern C++ Alignment (Cont…)

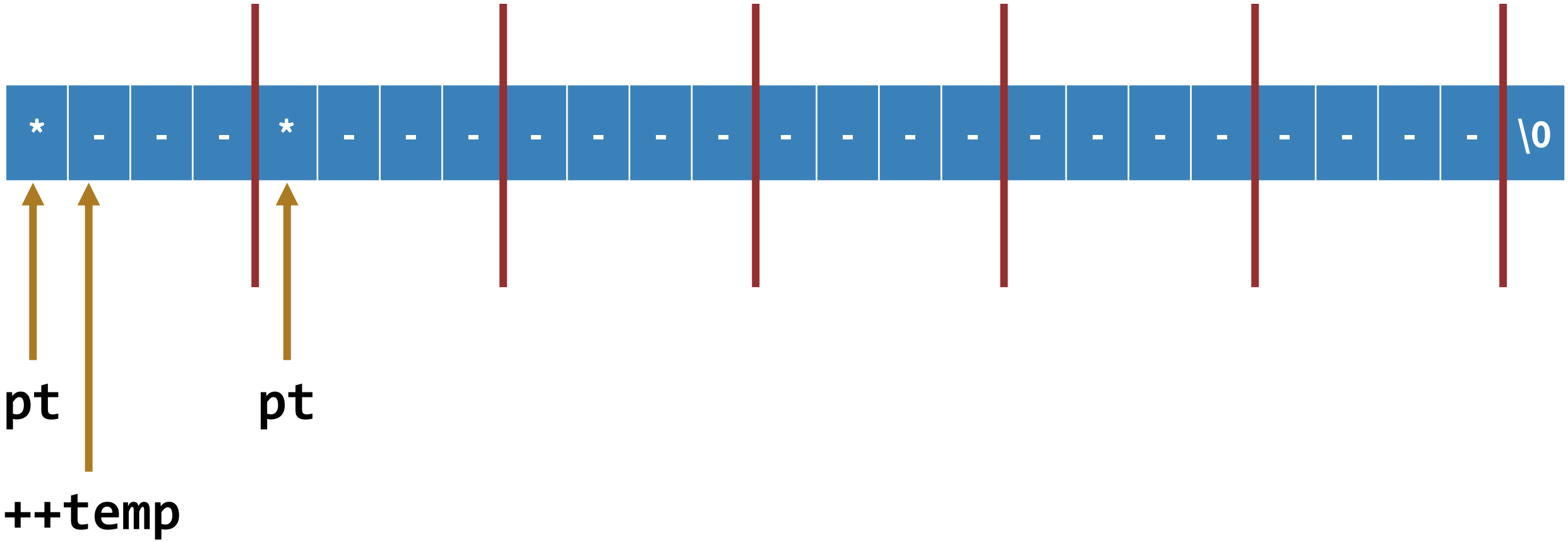
```
void* align(std::size_t req_alignment,
std::size_t size_in_bytes_to_align,
void*& returned_ptr,
std::size_t& returned_space_size);
```

If the aligned buffer fits into the proposed space, the function returns the updated value of *ptr*. Otherwise, a null pointer is returned.

Modern C++ Alignment (Cont…)

```
int main() {
    char buffer[] = "-----------------------";
    void * pt = buffer;
    std::size_t space = sizeof(buffer)-1;
    while ( std::align(alignof(int),sizeof(char),pt,space) ) {
        char* temp = static_cast<char*>(pt);
        *temp='*';
         ++temp;
        space-=sizeof(char);
        pt=temp;
    }
    std::cout << buffer << '\n';
    return 0;
}
```

Modern C++ Alignment (Cont…)



https://www.bfilipek.com/lmth.ngila-wenwen/2019/08

```
class alignas(32) Vec3d {
    double x, y, z;
};

int main() {
    auto Vec = Vec3d{};
    auto pVec = new Vec3d[10];

    if(reinterpret_cast<uintptr_t>(&Vec) % alignof(Vec3d) == 0)
        std::cout << "Vec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "Vec is not aligned to alignof(Vec3d)!\n";

    if(reinterpret_cast<uintptr_t>(pVec) % alignof(Vec3d) == 0)
        std::cout << "pVec is aligned to alignof(Vec3d)!\n";
    else
        std::cout << "pVec is not aligned to alignof(Vec3d)!\n";

    delete[] pVec;
}
```

Modern C++ Alignment (Cont…)

Let's try C++11/14 with GCC 4.8.5: (See @Wandbox):

```
sizeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is not aligned to alignof(Vec3d)!
```

And how about C++17, for example GCC 9.1 (see @Wandbox)

```
izeof(Vec3d) is 32
alignof(Vec3d) is 32
Vec is aligned to alignof(Vec3d)!
pVec is aligned to alignof(Vec3d)!
```

Modern C++ Alignment (Cont…)

- In C++11 and C++14, there was no guarantee that memory allocated for types that are over-aligned honors that specific alignment.

- In C++17, there is updated dynamic memory allocations, and now we have a guarantee that the memory will be aligned as requested.