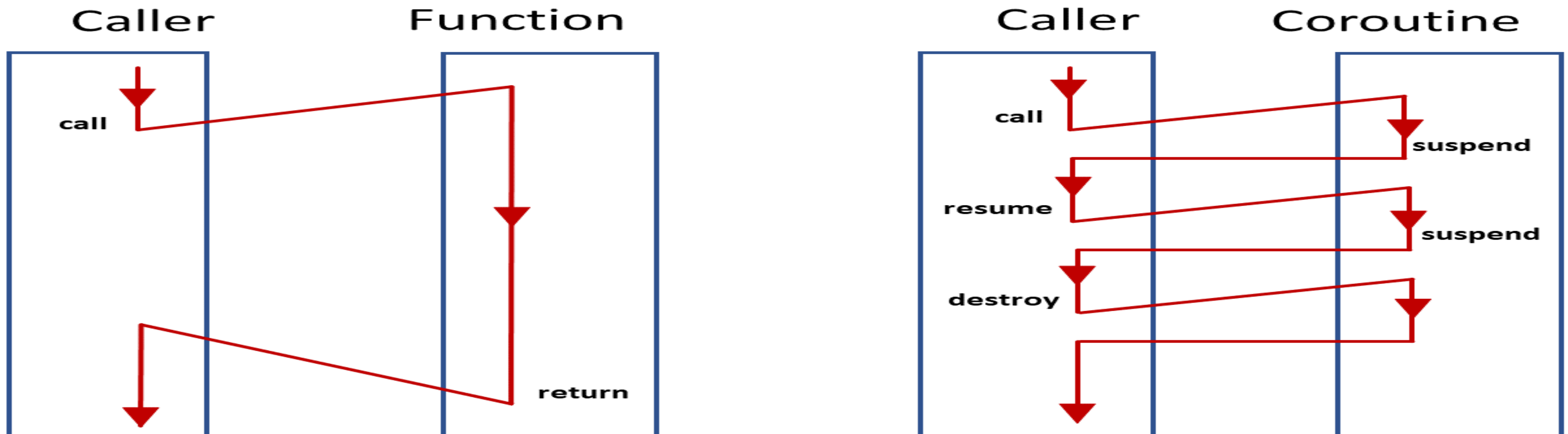
Coroutines (1)

If a function (caller) calls another function (callee), then the callee runs until its end or until *return* statement or until an exception is thrown. After that the local variables of callee are removed and the caller continues to run.

If the callee is not an ordinary function but a coroutine, the callee may temporarily suspend its work and send some results to the caller. The control is switched back to the caller and the callee pauses its work. When the caller has decided that the callee must continue its work (for example, when the analysis of data sent by callee is completed), it forces the callee to resume. It is not multithreading – the caller and callee do not run concurrently. We may say that the coroutine simply runs in background and suspends and resumes its work time to time. The figure is from https://www.modernescpp.com/index.php/implementing-futures-with-coroutines:



Coroutines (2)

A coroutine behaves like an object: it has state. Coroutines do not use the stack: on suspending the state is stored in a heap-allocated separate object.

A coroutine must contain at least one of the following keywords:

- co_await
- co_yield
- co_return

A suspended coroutine may simply wait for the resume command (*co_await*). It may also exchange data with the caller (*co_yield*) and at the end send to the caller some result (*co_return* but not *return*).

Formally, a coroutine is defined as any other function:

```
return_value_type coroutine_name(parameter_list) { body }
```

The return value type cannot be *void*, C++ standard type like *int* or standard class like *string*. It must be a user's class built according to some strictly specified rules. When the coroutine starts to run, an object of this class is created (so actually we do not have a return value as such). That object called as the coroutine interface is responsible for creating, running (i.e. suspending, resuming, data exchanging) and destroying of the associated with it coroutine.

The best source for studying coroutines is Nicolai M. Josuttis "C++ 20. The Complete Guide" 2022.

Coroutines (3)

In this example the coroutine reads row-by-row text from a file. When a row is read, the coroutine prints it and suspends. The human operator reads the text and resumes the reading or ends the work.

```
#include <coroutine>
ReturnType coro(fstream & f) { // coroutine interface here is an object of class ReturnType
  char line[256];
  while (true) {
      f.getline(line, 256);
      if (f.eof()) {
           cout << "No more data" << endl;
           break;
      }
      cout << "coro  has read text: " << line << endl;
      co_await suspend_always{}; // Suspend point
  // co_await is an operator, its operand is an object specifying the behavior of suspension
  // Here we use nameless object from standard class suspend_always
  // Object from class suspend_always switches the control back to caller
  }
 // no return expression here, the control is switched back to caller
}
```

Coroutines (4)

```
void TestCoroutine () {  // testing the coroutine from the previous slide
  fstream File;
  File.open("sometext.txt",  fstream::in); // open the file
  if (!File.good()) {
      cout << "Failed to open file" << endl;
      return;
  }
  ReturnType task = coro(File); // Initialize and launch the coroutine
                                // Object task is the coroutine interface
  while (true)  {
      cout << "Press a key to continue, ESC to stop" << endl;
      if (_getch() == 27) {
            break;  // ESC - stop reading from file
      }
     if (!task.resume()) { // When coro has suspended its work, resume it after keystroke
            break;  // If resume() returns false, the coroutine has exited
      }
  }
  File.close();
}
```

Coroutines (5)

The *ReturnType* class (the class name may be, of course, any) must contain:

```
class ReturnType {
public:
struct promise_type; // declaration on the next slide
// Nested definition of a class that presents the coroutine promise (this is not the promise
// used in multithreading but its function is similar: the coroutine submits the results and
// exceptions through the promise object. Traditionally, the promise is defined as "struct
// promise_type".
coroutine_handle<promise_type> handle;
// The coroutine handle, this must be an object of class coroutine_handle<promise_type>
ReturnType(coroutine_handle<promise_type> h) : handle(h) { } // Constructor
~ReturnType() { handle.destroy(); } // Destructor
// method coroutine_handle<promise_type>::destroy() destroys the coroutine
bool resume() const {  // Function resume(), see the usage from the previous slide
   if (!handle || handle.done()) {
// if the coroutine has exited, method coroutine_handle<promise_type>::done() returns false
       return false;
   }
   handle.resume(); // coroutine_handle<promise_type ::resume() resumes the coroutine
   return !handle.done();
  }
};
```

Coroutines (6)

```
struct ReturnType::promise_type { // the following methods are mandatory
auto get_return_object() {
	return ReturnType{ coroutine_handle<promise_type>::from_promise(*this) }; }
// Called to create and initialize the coroutine inteface. Static method
// coroutine_handle<promise_type>::from_promise() creates handle from the current
// promise, this handle is used as argument of the ReturnType constructor
auto initial_suspend() { return suspend_always(); }
// If this function returns object from standard class suspend_always, the coroutine starts
// lazily, i.e. it is suspended at the very beginning (although there is no suspend point). If the
// return value is of class suspend_never, the coroutine runs until the first suspend point
// (starting eagerly). See comments on the next slide.
void unhandled_exception() { terminate(); }
// How to deal with exceptions unhandled by the coroutine. Complicated problem, here we
// simply terminate the program
void return_void() { }
// As in our example the coroutine returns nothing and does not yields data, this function
// must be empty
auto final_suspend() noexcept { return suspend_always(); }
// Similar to initial_suspend(): defines wether the coroutine must be suspended at the end
};  // end of struct promise_type
```

Coroutines (7)

Comments:

auto initial_suspend() { return suspend_always(); } means that the coroutine stops at the beginning. The caller runs until the point where *resume()* is called. There it stops and the coroutine starts to run. It runs until the suspend point.

auto initial_suspend() { return suspend_never(); } means that the coroutine starts to run immediately and the caller is blocked. The coroutine stops at the suspend point. After that the caller ismunblocked and runs until the point where *resume()* is called.

Coroutines (8)

In the following example the coroutine reads row-by-row text from a file, converts into string and yields to the caller:

```
#include <coroutine>
ReturnType coro(fstream & f) { // coroutine interface here is an object of class ReturnType
  char line[256];
  while (true) {
      f.getline(line, 256);
      if (f.eof()) {
          cout << "No more data" << endl;
          break;
      }
     co_yield string(line); // Suspend point
                            // co_yield is an operator, its operand is the value to send to the caller
  }
 // no return expression here, the control is switched back to caller
}
```

Coroutines (9)

```
void TestCoroutine () {  // testing the coroutine from the previous slide
  fstream File;
  File.open("sometext.txt",  fstream::in); // open the file
  if (!File.good()) {
      cout << "Failed to open file" << endl;
      return;
  }
  ReturnType task = coro(File); // Initalize and launch the coroutine
                                // Object task is the coroutine interface
  while (true)  {
      cout << "Press a key to continue, ESC to stop" << endl;
      if (_getch() == 27) {
            break;  // ESC - stop reading from file
      }
     if (!task.resume()) { // When coro has suspended its work, resume it after keystroke
            break;  // If resume() returns false, the coroutine has exited
      }
      cout << task.get_value() << endl; // prints the value got from coroutine
}
  File.close();
}
```

Coroutines (10)

The *ReturnType* class has some new members (compare with slides *Coroutines(5)* and *(6)*):

```
class ReturnType {
public:
    struct promise_type {
        string coro_value = string("");  // recent value from co_yield
        auto yield_value(string s) { // reaction to co_yield
            coro_value = s;
            return suspend_always { };
        }
        ………………………….. // as on slide Coroutines (6)
    };
    ……………………………….  // as on slide Coroutines (5)
    string get_value() const {
        return handle.promise().coro_value;
    }
};
```

Coroutines (11)

The difference between this example and the example on slide *Coroutines (3)* is that at the end of its work the coroutine returns the number of rows it has read.

```
#include <coroutine>
ReturnType coro(fstream & f) { // coroutine interface here is an object of class ReturnType
  char line[256];
  int nRow = 0;
  while (true) {
      f.getline(line, 256);
      if (f.eof()) {
           cout << "No more data" << endl;
           break;
     }
     cout << "coro  has read text: " << line << endl;
     nRow++;
     co_await suspend_always{}; // Suspend point
    }
  co_return nRow;
}
```

Coroutines (12)

```
void TestCoroutine () {  // testing the coroutine from the previous slide
  fstream File;
  File.open("sometext.txt",  fstream::in); // open the file
  if (!File.good()) {
      cout << "Failed to open file" << endl;
      return;
  }
  ReturnType task = coro(File); // Initalize and launch the coroutine
                                // Object task is the coroutine interface
  while (true)  {
      cout << "Press a key to continue, ESC to stop" << endl;
      if (_getch() == 27) {
          return;  // ESC - stop reading from file
      }
     if (!task.resume()) { // When coro has suspended its work, resume it after keystroke
          break;  // If resume() returns false, the coroutine has exited
      }
  }
  cout << task.get_result() << endl;
  File.close();
}
```

Coroutines (13)

The *ReturnType* class has some new members (compare with slides *Coroutines(5)* and *(6)*):

```
class ReturnType {
public:
    struct promise_type {
        int coro_result;   // value from co_return
        void return_value(const int & i) {  // reaction to co_return
            coro_result = i;
        }
        ……………………….. // as on slide Coroutines (6)
                              // void return_void() { } must be removed
    };
    …………………………….  // as on slide Coroutines (5)
    string get_result() const {
        return handle.promise().coro_result;
    }
};
```

Coroutines (14)

Objects that can be operands for operator *co_await* are called as awaiters. C++ has two standard awaiters classes: *suspend_always* and *suspend_never*. Rather often a programmer has to write his / her own awaiter class.

An awaiter class must contain the following methods:

1. Method *await_ready()* is called by supporting system right before the coroutine is suspended. Return *value* true means that the suspension must be ignored, *false* means that the suspension is allowed. In class *suspend_always* this function is:

```
bool await_ready() const noexcept { return false; }
```

2. Method *await_suspend(coroutine_handle<>)* is called by supporting system right after the coroutine is suspended. Its argument is the coroutine handle. In class *suspend_always* this function is empty:

```
void await_suspend(coroutine_handle<void> h) noexcept { } // template has no argument
```

await_suspend() may have return value. With this function you can specify what to do next.

3. Method *await_resume()* is called by supporting system right after the coroutine has resumed its work. In class *suspend_always* this function is empty:

```
void await_resume() const noexcept { }
```

await_resume() may have a return value. With this method we can specify the value that the caller sends back to coroutine

Coroutines (15)

In this example the coroutine reads row-by-row text from a file, converts into string and yields to the caller. The caller decides wether to continue or quit the coroutine and sends a bool value back to the coroutine..

```
#include <coroutine>
ReturnType coro(fstream & f) { // coroutine interface here is an object of class ReturnType
  char line[256];
  bool run = true;
  while (run) {
      f.getline(line, 256);
      if (f.eof()) {
           cout << "No more data" << endl;
           break;
      }
    run = co_yield string(line); // Suspend point
                              // co_yield is an operator, its operand is the value to send to the caller
                              // the result of the co_yield operation is the value delivered by the caller
  }
 // no return expression here, the control is switched back to caller
}
```

Coroutines (16)

```
void TestCoroutine () {  // testing the coroutine from the previous slide
  fstream File;
  File.open("sometext.txt",  fstream::in); // open the file
  if (!File.good()) {
      cout << "Failed to open file" << endl;
      return;
  }
  ReturnType task = coro(File); // Start the coroutine, task is the coroutine interface
  while (true)  {
     cout << "Press a key to continue, ESC to stop coroutine" << endl;
     if (!task.resume())
           break;  // If resume() returns false, the coroutine has exited
     }
      cout << task.get_value() << endl; // prints the value got from coroutine
     _getch() == 27 ? task.set_back_value(false) : task.set_back_value(true);
     }
  File.close();
}
```

Coroutines (17)

The *ReturnType* class has some new members (compare with slides *Coroutines(5)* and *(6)*):

```
class ReturnType {
public:
    struct promise_type {
        string coro_value = string("");  // recent value from co_yield
        bool back_value = true;
        auto yield_value(string s) { // reaction to co_yield
            coro_value = s;
            back_value = true;
            return back_awaiter<coroutine_handle<promise_type>> {  };
        }
        ………………………….. // as on slide Coroutines (6)
    };
    ……………………………….  // as on slide Coroutines (5)
    string get_value() const {
        return handle.promise().coro_value;
    }
    void set_back_value(const bool& b) {
        handle.promise().back_value = b;
    }
};
```

Coroutines (18)

```
template<typename HANDLE> class back_awaiter
{
public:
    HANDLE handle = nullptr;
    back_awaiter() = default;
    bool await_ready() const noexcept { return false; }
    void await_suspend(HANDLE h) noexcept { handle = h;  }
      // Here we just store the value of handler as the attribute of awaiter
     auto await_resume() const noexcept { return handle.promise().back_value; }
      // Here we can specify the value that the caller sends back to coroutine
};
```

Ranges (1)

As an abstraction, a range is something that you can iterate over. A range is represented by an iterator that marks the beginning of the range and a sentinel that marks the end of the range. The sentinel may be the same type as the begin iterator, or it may be different.

All the STL containers (vectors, lists, maps, etc.) are ranges. Their iterators marking the beginning and the end are of the same type.

```
vector<int> v { 67, 34, 78, 1, 14 };
sort(v.begin(), v.end()); // get sorted vector
```

There is also a standard function for sorting that uses ranges:

```
#include <ranges>
ranges::sort(v); // algorithms from the ranges library are in their own namespace
```

Most (but not all) of the STL standard algorithms (see chapter *"Algorithms"*) have equivalents from the ranges library. Another example:

```
auto print = [](const int &i) { cout << i << ' ';  }; // lambda
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
for_each(data.begin(), data.end(), print);
cout << endl;
ranges::for_each(data, print); // more comfortable
cout << endl;
```

In Visual Studio 2022 set the C++ language standard to *Preview – features from the latest C++ working draft*.

Ranges (2)

Applying the view adapters, we can from ranges create views. A view is also range and consequently can be the argument for an range library function . Examples of view adapters:

```
ranges::for_each(views::all(data), print); // the view contains the complete range
cout << endl;  //  result: 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7
ranges::for_each(views::take(data, 2), print); // the view contains the first two elements of
// range
cout << endl; //  result: 1, 4
ranges::for_each(views::drop(data, 2), print); // the view contains all the elements of range
// except the first two
cout << endl; //  result: 5, -7, 3, -8, 9, 12, 56, -45, 7
ranges::for_each(views::filter(data,[](const int& i) { return i > 0; }), print); // the view
// contains all the positive elements of range
cout << endl; //  result: 1, 4, 5, 3, 9, 12, 56, 7
ranges::for_each(views::take_while(data, [](const int& i) { return i > 0; }), print); // the view
// contains elements until the first negative value
cout << endl; //  result: 1, 4, 5
```

Constructing a view does not perform any operations on the range, i.e. a view does not own any elements. The operations on a view are applied only when the iterating over the elements is going on.

The complete list of view adapters is on https://en.cppreference.com/w/cpp/ranges

Ranges (3)

As the views are ranges, we can use them in range-based *for* loops (see chapter *Containers*):

```
for (const auto& i : data) {
    cout << i << ' ';
}
cout << endl; //  result: 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7
for (const auto& i : views::filter(data, [](const int& i) { return i > 0; })) {
    cout << i << ' ';
}
cout << endl; // result: 1, 4, 5, 3, 9, 12, 56, 7 (negative values are filtered out)
```

There is an alternative syntacs:

```
for (const auto& i : data | views::filter([](const int& i) { return i > 0; })) {
    cout << i << ' ';
}
cout << endl; // result: 1, 4, 5, 3, 9, 12, 56, 7
```

A view can have name:

```
auto view1 = views::filter(data, [](const int& i) { return i > 0; });  // no any filtering here
```

or

```
auto view2 = data | views::filter([](const int& i) { return i > 0; });
for (const auto &i : view1) {
     cout << i << ' ';  // filtering happens here
}
```

Ranges (4)

Ranges and views can be connected into pipeline:

```
for (const auto& i : data | views::filter([](const int& i) { return i > 0; }) |
                            views::filter([](const int& i) { return i % 2;  } )) {
    cout << i << ' ';
}
cout << endl; // result: 1, 5, 3, 9, 7 (only positive and odd numbers)
```

Turn attention that with view adapter *transform* you cannot store the changes:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto view3 = views::transform(data, [](const int& i) { return i > 0 ? i * 2 : i; });
ranges::for_each(view3, print);
cout << endl; // results: 2 8 10 -7 6 -8 18 24 112 -45 14 (positive values multiplied by 2)
ranges::for_each(data, print);
cout << endl; //  results still: 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7
```

The following code works:

```
for (auto& i : view2) { // view2 contains positive elements
     i *= 2;
}
ranges::for_each(data, print);
cout << endl; // results: 2 8 10 -7 6 -8 18 24 112 -45 14
```

Ranges (5)

However, there is a possibility to get a container from a range or view:

```
vector<int> data = { 1, 4, 5, -7, 3, -8, 9, 12, 56, -45, 7 };
auto view4 = views::transform(data, [](const int& i) { return i > 0 ? i * 2 : i; });
vector<int> new_data(view4.begin(), view4.end()); // define new vector
for_each(new_data, print);
cout << endl; // results: 2 8 10 -7 6 -8 18 24 112 -45 14
```

Several functions from the ranges library have an additional input parameter. Example:

```
struct Student {
string first_name, last_name;
Student(string s1, string s2) : first_name(s1), last_name(s2) { } };

vector<Student> group{ { "John", "Smith"}, { "Mary", "Clerk"}, { "James", "Sailor"} };
ranges::sort(group); // error, does not compile because Student does not have operator<
```

We need to use comparator explaining how to compare vector elements:

```
ranges::sort(group, [](Student s1, Student s2) { return s1.last_name < s2.last_name; });
ranges::for_each(group, [](Student s) { cout << s.first_name << ' ' << s.last_name << endl; });
// get Mary Clerk, James Sailor, John Smith
```

But we may also use a projection: a lambda, function or functor that modifies the values coming from the container and pass the projected values to the algorithm:

```
ranges::sort(group, { }, [](Student s) { return s.first_name; }); // from struct to string
ranges::for_each(group, [](Student s) { cout << s.first_name << ' ' << s.last_name << endl; });
// get James Sailor, John Smith, Mary Clerk
```

Modules (1)

Header files (*.h* files) have several problems. For example, we need to avoid multiple including of the same file, sometimes we must watch on the order of *#include* directives, etc. The standard header files may consist of tens of thousands of lines of code – this is a problem for compilers. Modules are to solve those issues.

To work with modules in Visual Studio 2022 you need to install the modules for version build tools. To see the instruction type into Google "error C1011" (if the mentioned tools are not installed you get this fatal error).

Also, set the C++ language standard to *Preview – features from the latest C++ working draft (std:c++ latest)* and Enable Experimental C++ Standard Library Module to *Yes(/experimental:module)*.

Do not try to insert modules into a half ready project. Use modules as the main building components in your next project that you will start from scratch.

Caution: actually the support of modules in Visual Studio 2022 is experimental (although Microsoft claims that C++ v.20 standard modules are fully implemented). You may get strange error messages or even message E1504 (internal compiler error).

Some useful links:

https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170
https://www.modernescpp.com/index.php/cpp20-a-first-module
https://www.modernescpp.com/index.php/c-20-module-interface-unit-and-module-implementation-unit

Modules (2)

A module consists of one (only one) module interface file and optional number of module implementation files. In Visual Studio 2022 the extension of module interface file is **.ixx*. Some other compilers use extension **.cppm*. The filename may be any. The module implementation files are ordinary *.cpp files.

To start add to your project a new module interface file (in Visual Studio it is similar to adding a new item or class). Example:

```
module; // starts to declare a module
// The next block is the global module fragment. Mostly it contains #include directives
// non-importable header files like *.h files from C
#include "stdio.h"
// Then we must set the module name. By default it is the same as the interface filename
export module Example1;
// In the next block we specify modules that our module must import. The C++ include files
// are importable. We may also import other modules from our project.
import <iostream>;
import <numbers>;
import <string>;
// Next the non-exported declarations must follow
using namespace std;
```

Modules (3)

```
// At last we must specify what our module will export. We may export classes, structs,
// functions, namespaces, enumeration classes, etc.
export class Circle
{
public:
      double radius = 0,
                  area = 0,
                  perimeter = 0;
      Circle(double r) : radius(r) {
            area = numbers::pi * radius * radius;
            perimeter = 2 * numbers::pi * radius;
      }
    string ToString()  {
          return "Radius: " + to_string(radius) + "  Area: " + to_string(area) +
                                 " Perimeter: " + to_string(perimeter);  }
     void PrintCircle() {
           printf("%s\n", ToString().c_str());  }
};
```

Here is the end of module intereface. In this simple example module implementation files are not needed.

Modules (4)

Here is file testing module Example1:

```
import <iostream>;
import Example1;
using namespace std;
int main()  {
   Circle c(10);
   cout << "Circle parameters ";
   c.PrintCircle();
   return 0;
}
```

Turn attention that although we imported *iostream* into module *Example1*, in the file for testing the module (this file does not belong to the module) we need to import it once more. *iostream* and any other module imported into our *Example1* module are not automatically exported into into files that in turn import *Example1*.

Instead of importing the C++ standard headers separately, you may import them with one sentence:

```
import std.core;
```

std.core does not include headers *<atomic>*, *<condition_variable>*, *<future>*, *<mutex>*, *<thread>*, they are joined into *std.threading*. See more from https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170

21.06.23 – there were problems with std.core

Modules (5)

```
module;  // module interface file Example2.ixx
#include "time.h"
…………………………… // other header files
export module Example2;
import <iostream>;
…………………………….. // other imported modules
using namespace std;
export class Date {
private:
    int Day;
    char Month[4];
    int Year;
    …………………………. // other attributes
public:
    Date();
    Date(int d, int m, int y);
    Date(const Date&);
    …………………………….. // other methods
};
export Date CreateRandomDate(Date begin, Date end);   // exported function
```

Modules (6)

```
#include "time.h"  // needed because time() is a static function
module Example2; // module implementation file Date.cpp
#pragma warning(disable : 4996)
Date::Date() {  // constructor
   time(&Now);
   struct tm Tm;
   localtime_s(&Tm, &Now);
   Day = Tm.tm_mday; // 1...31
   strcpy(Month, MonthNames[Tm.tm_mon]); // 0...11
   iMonth = Tm.tm_mon + 1;
   Year = Tm.tm_year + 1900; // current year - 1900
}
……………………………….. // other methods
Date CreateRandomDate(Date begin, Date end)
{  // implementation of function declared in module interface file
  ……………………………….
}
```

Modules (7)

If you do not use modules and make changes in a **.h* file, the system recompiles the complete project. Therefore, if you want to save time, put the code of class methods not into the class declaration but into a separate *.cpp file. In module files making changes in the code of class methods does not require the immediate recompilation (exception: it is true if you do not touch the function header). This is because a module interface file does not include any function implementations, even if the implementations are directly written into the interface file (as they are in *Example1*). In other words, methods implemented in *.h files are by default inline methods, methods implemented in modules are not.

Formatted output (1)

Let us have

```
struct invoice { const char* pPassenger, * pDestination; const int fare; };
invoice InvoiceList[] = {
 { "John Smith", "Quito", 2000 },
 { "Richard Carpenter", "Guatemala City", 1700 } };
```

If we print it out using code snippet:

```
auto print = [](const invoice& inv) { cout << inv.pPassenger << ' '<< inv.pDestination << ' '
                                      << inv.fare << endl;  };
ranges::for_each(InvoiceList, print);
```

we get:

```
John Smith Quito 2000
Richard Carpenter Guatemala City 1700
```

To get a correct table we need better formatting:

```
auto print = [](const invoice& inv) { cout << setw(17) << left << inv.pPassenger << ' '
                                      << setw(14) << inv.pDestination << ' ' <<
                                      inv.fare << endl;  };
```

```
John Smith        Quito          2000
Richard Carpenter Guatemala City 1700
```

The code of the last lambda is rather inconsiderate and clumsy.

Formatted output (2)

We may also use *printf*:

```
for (int i = 0; i < 2; i++) {
     printf("%-17s %-14s %d\n", InvoiceList[i].pPassenger, InvoiceList[i].pDestination,
             InvoiceList[i].fare);
}
```

This code is more comprehensive, but it is C, not C++.

Formatting in C++ v. 20 is a combination of *iostream* and *printf*:

```
#include <format> // see https://en.cppreference.com/w/cpp/utility/format
auto print = [](const invoice& inv) { cout << format("{:17} {:14} {}\n", inv.pPassenger,
                                        inv.pDestination, inv.fare); };
ranges::for_each(InvoiceList, print);
```

As *printf*, *format* has formatting string followed by the sequence of arguments. The formatting rules for each argument are enclosed into braces. If there are no any specific requirements, the braces are empty and *format* deduces the type itself:

```
cout << format("{}\n", arg); // arg may be int, double, C-style string, C++ string object, etc.
```

Do not put spaces between braces, you will get an error message!

The order of formatting rules in formatting string and arguments in list may not match:

```
cout << format("{1:17} {0:14} {2}\n", inv.pPassenger, inv.pDestination, inv.fare);
```

Now the printout starts with the destination, the passenger's name will follow. Number right after *{* specifies the order.

Formatted output (3)

Formatting rules start with colon.

A positive integer after colon specifies the field width:

```
cout << format("{:6}\n", 10);  // prints ␣␣␣␣10 (by default right alignment)
cout << format("{:<6}\n", 10); // prints 10␣␣␣␣
cout << format("{:^6}\n", 10); // prints ␣␣10␣␣
cout << format("{:6}\n", "ab");  // prints ab␣␣␣␣ ((by default left alignment)
cout << format("{:6}\n", "abcdefghijkl");  // prints the complete string abcdefghijkl
```

Space is the default filling character. But you can use another:

```
cout << format("{:_<6}\n", 11); // prints 11____
cout << format("{:0^6}\n", 11); // prints 001100
cout << format("{:_>6}\n", 11); // prints ____11, but {:_6} is an error
```

The precision is used for floating-point values and strings:

```
cout << format("{}\n", 0.123456789); // prints 0.123456789
cout << format("{:.5}\n", 0.123456789); // prints 0.12346 (rounds up)
cout << format("{:^11.5}\n", 0.123456789); // prints ␣␣0.12346␣␣
cout << format("{:.5}\n", "abcdefghijk"); // prints abcde
cout << format("{:>10.2}\n", "abcdefghijk"); // prints ␣␣␣␣␣␣␣␣ab
```

Formatted output (4)

Type specifiers for integral types:

```
cout << format("{:d} {:s}\n", true, true); // prints 1 true
cout << format("{:d} {:x} {:X} {:c}\n", 'Z', 'Z', 'Z', 'Z'); // prints 90 5a 5A Z
cout << format("{:#X} {:b} {:#B}", 251, 251, 251); // prints 0XFB 11111011 0B11111011
```

Type specifiers for floating-point types:

```
cout << format("{} {:f} {:g} {:E}\n", -5.0, -5.0, -5.0, -5.0);
   // prints -5 -5.000000 -5 -5.000000E+00
cout << format("{} {:f} {:g} {:e}\n", -5.5, -5.5, -5.5, -5.5);
   // prints -5.5 -5.500000 -5.5 -5.500000e+00
cout << format("{} {:f} {:g} {:e}\n", numbers::pi, numbers::pi, numbers::pi * 1e-3,
                    numbers::pi * 1e6);
   // prints 3.141592653589793 3.1241593 0.00314159 3.14159e+06
```

It is possible to store the result into a buffer:

```
char buf1[256];
auto ret1 = format_to_n(buf1, size(buf1) - 1, "{}", 100);
*(ret1.out) = 0;  // we must ourself add the terminating zero
```

or

```
array<char, 256> buf2{};  // fills with zeroes
auto ret2 = format_to_n(buf2.begin(), size(buf2) - 1, "{}", 100);
```

Formatted output (5)

To calculate the needed size of buffer:

```
auto n = formatted_size("{}", 100);  // returns 3, so the actual size is n + 1
```

To work without predefined buffer:

```
string s;
format_to(back_inserter(s), "{}", 100);  // see "Insert iterators" from chapter "Algorithms"
format_to(back_inserter(s), "{}", ' ');
format_to(back_inserter(s), "{}", 200);
cout << s << endl; // prints 100 200, tghus we may replace the stringstream
```

The format string must be a compile-time value:

```
const char *pfr = "{}\n";
cout << format(pfr, 100); // error
constexpr const char *pfr = "{}\n";
    // correct, see "Constant expressions" from chapter "Advanced C++"
```

However, sometimes we may need to create the formatting string at runtime. In that case we can use *vformat* and *vformat_to* together with function *make_format_args*:

```
const char *pfr = "{} {:f} {:g} {:e}\n";
cout << vformat(pfr, make_format_args(numbers::pi, numbers::pi, numbers::pi * 1e-3,
                         numbers::pi * 1e6);
```

Comparisons (1)

Let us have a simple class:

```
class Id {
    long long int id = 0;
public:
    Id() {  }
    Id(long long int l) : id(l) { };
    long long int get() const { return id; }
    void set(long long int l) { id = l; }
};
```

To check are two objects from class *Id* equal or not we have to add to operator functions:

```
bool operator==(const Id &i) const { return id == i.get(); }
bool operator!=(const Id &i) const { return id != i.get(); }
```

Now

```
Id nr1(77LL), nr2(77LL), nr3(88LL);
cout << boolalpha << (nr1 == nr2) << ' ' <<  (nr1 == nr3) << ' ' << (nr1  != nr3) << endl;
                                                                          // prints true false true
```

For the complete set of comparisons we need four more operator functions: *operator>*, *operator<*, *operator>=* and *operator <=*.

If we want to compare the value in object with an integer we need another operator function

```
bool operator==(const long long int &l) const { return id == l;}
```

Comparisons (2)

If we want to compare the value in object with an integer we need another operator function:

```
bool operator==(const long long int &l) const { return id == l; }
```

Now:

```
cout << boolalpha <<  (nr1 == 99LL) << endl; // prints false
```

but

```
cout << boolalpha <<  (99LL == nr1) << endl; // error
```

Consequently we need one more operator function:

```
friend bool operator==(const long long int& l, const Id& i) { return l == i.id;  }
```

Totally, we may need for class *Id* 18 operator functions.

However, in C++ v. 20 if *operator==* is implemented, *operator!=* is not needed, the inequality can be checked without it. Also, it is enough to impelement one operator== for comparison of objects of different types. So

```
class Id {
……………………
bool operator==(const Id &i) const { return id == i.get(); }
bool operator==(const long long int &l) const { return id == l; }
};
Id nr1(77LL), nr2(77LL), nr3(88LL);
cout << format("{:s} {:s}\n", nr1 == nr2, nr1 != nr3);  // prints true true
cout << format(("{:s} {:s}\n", , nr1 == 99LL, 99LL != nr1); // prints false true
```

Comparisons (3)

But if we have only *operator>*,

```
cout << format("{:s} {:s}\n", nr1 > nr2, nr1 < nr3); // error, operator< is also needed
```

The solution is to use new C++ v. 20 operator<=> (three-way comparison operator, called also as spaceship).

Three-way comparison can be used for primitive types:

```
#include <compare> // https://en.cppreference.com/w/cpp/utility/compare/compare_three_way
int i1 = 10, i2 = 10, i3 = 20;
auto ret1 = i1 <=> i2;  // the return value cannot be bool
cout << typeid(ret1).name() << endl;  // prints struct std::strong_ordering
auto ret2 = i2 <=> i1;
auto ret3 = i1 <=> i3;
auto ret4 = i3 <=> i1;
auto iprint = [](auto ret)->string {
                    if (ret == strong_ordering::less) return "less";
                    else if (ret == strong_ordering::greater) return "greater";
                    else return "equal";  }; // strong_ordering::equal
cout << format("{} {} {} {}\n", iprint(ret1), iprint(ret2), iprint(ret3), iprint(ret4));
   // prints equal equal less greater
```

strong_ordering is something similar to enumeration but cannot be used in *switch* statements. For floating-point values the result type is *partial_ordering*, it includes also member *unordered* (for situation in which one of the operands is not a number).

Comparisons (4)

All the primitive types as well as all the C++ standard classes for which the relational operations are defined support also the three-way comparison. Of course, there is no sense to use the spaceship operator for comparison of primitive types.

The main strength of three-way comparison is that we may in our own classes replace *operator<*, *operator<=*, *operator>* and *operator>=* functions with one function *operator<=>*:

```
auto operator<=>(const Id& i) const { return id <=> i.get(); }
```

Now:

```
Id nr1(77LL), nr2(77LL), nr3(88LL);
auto ret1 = nr1 <=> nr3;
auto ret2 = nr3 <=> nr1;
auto ret3 = nr1 <=> nr2;
cout << format("{} {} {}\n", ret1._Value, ret2._Value, ret3._Value); // prints -1 1 0
```

If the _*Value* is negative, the first operand is less, if positive, the first operand is greater and if 0, they are equal (as in standard function *strcmp*).

Three-way comparison result may be checked by standard functions *is_eq*, *is_neq*, *is_lt*, *is_lteq*, *is_gt*, *is_gteq*:

```
cout << format("{} {} \n", is_eq(nr1<=>nr3), is_gt(nr3<=>nr1)); // prints false true
```

Comparisons (5)

C++ v. 20 compiler is able to generate default *operator*== and *operator* <=> functions:

```
class Name {
    string FirstName = "", LastName = "";
public:
    Name(string s1, string s2) : FirstName(s1), LastName(s2) { }
    bool operator==(const Name &n) const = default;
    auto operator<=>(const Name &n) const = default;
};

Name p1("Clyde", "Barrow"), p2("Clyde", "Barrow"), p3("Bonnie", "Parker");
cout << format("{:s} {:s} {:s}\n", p1 == p2, p1 == p3, p1 != p3);
            // prints true false true
cout << format("{} {} {}\n", (p1 <=> p2)._Value, (p1 <=> p3)._Value, (p3 <=> p1)._Value);
            // prints 0 1 -1
```

As in the default copy constructor, default comparison functions perform the operations attribute by attribute. If some of the attributes are pointers, it does not work and we need to write our comparison ourselves.

Multithreading (1)

Let us have a simple thread:

```
void Test () {
    int n = 10;
    thread Worker([&]() { for (int i = 0; i < n; i++) {
                              this_thread::sleep_for(chrono::seconds(1));
                              cout << i << endl;
                          }  // thread body is a lambda
                          cout << "Ready";  });
   ................................... // do something
    return;  // here the Worker object is destroyed
}
```

If the thread object is destroyed when the thread is still running, the program will creash. The solution is to use method *join*:

```
Worker.join(); // blocks function Test() until the end of thread
```

The problem here is that function *Test* may have several *return* amd *throw* points, so we need to call *join* in many places. If, in addition, we have more than one thread, we may get very complicated and indistinct code.

If we instead of class *thread* apply C++ v. 20 class *jthread*, method *join* is called automatically by the destructor of this class and the code presented on this slide will work.

Multithreading (2)

```
void Test () {
  ........................... // Do something
  jthread Worker(/* entry point function and input parameters */) ;
   ........................... // Do something
   if (/* some condition */ )  {
     return;  // no need to call join
   try {
         ........................... // Do something
   }
    catch (exception) {
      return; // no need to call join
   }
 Worker.join();
 ........................... // the thread has finished, analyse the results
}
```

jthread provides the same interface as *thread* so it is possible in old code simply add character j to classname *thread*. Header file <*thread*> presents the both classes.

Multithreading (3)

Rather often a thread runs in endless loop and we need a mechanism to interrupt it. In C++ killing of thread is not allowed. What we need are tools for so called cooperative (sometimes said polite) cancellation with which we force the thread entry point function to return.

On slides *Atomic variable (4)*, *Condtional variables (8)…(10)* from chaper *"Concurrency"* we solved this proble with a flag – an *atomic<bool>* global variable. The thread entry point function checks periodically this flag and quits if an outer function has set it to *false*. Class *jthread* has better tools:

```
#include <stop_token> // see https://en.cppreference.com/w/cpp/header/stop_token
void Worker(int n, stop_token stop) { // obligatory parameter of type stop_token
  for (int i = 0; i < n && !stop.stop_requested(); i++) {
    this_thread::sleep_for(chrono::seconds(1));
    cout << i << endl;
  }   // the thread must time to time check the state of stop_token object
}
void Test {
  stop_source source;   // to control the interrupting
  stop_token stop = source.get_token();
  jthread thr(Worker, 10, stop);
  this_thread::sleep_for(chrono::seconds(5));  // allow to run 5 seconds and then interrupt
  source.request_stop();  // after that stop_requested() in Worker returns true
}
```

Multithreading (4)

It is possible to associate the stop_token object with a callback (function, functor or lambda). The callback is invoked when the stop is requested:

```
void Worker(int n, stop_token stop)
{
   stop_callback OnRequest(stop, []() { cout << "Broken" << endl; });
   for (int i = 0; i < n && !stop.stop_requested(); i++)
  {// the thread must time to time check the state of stop_token object
      this_thread::sleep_for(chrono::seconds(1));
      cout << i << endl;
   }
}
void Test
{
   stop_source source;   // to control the interrupting
   stop_token stop = source.get_token();
   jthread thr(Worker, 10, stop);
   this_thread::sleep_for(chrono::seconds(5));
   source.request_stop();  // after that stop_requested() in Worker returns true
                           // in addition callback OnRequest is invoked,
                           // i.e. message "Broken" is printed
}
```

Multithreading (5)

It may happen that when the stop is requested, a thread is waiting for notification for a *condition_variable* and cannot check the state of *stop_token.* The following example presents the solution of this problem:

```
void Test {
   mutex mx;
   condition_variable_any cv;  // use instead of condition_variable
                                // see https://en.cppreference.com/w/cpp/thread/condition_variable_any
   queue<int> values;
   stop_source source;   // to control the interrupting
   stop_token stop = source.get_token();
   jthread thr1(Consumer, &mx, &cv, &values, stop);  // on the next slides
   jthread thr2(Producer, &mx, &cv, &values, stop);
   this_thread::sleep_for(chrono::seconds(5)); // allow to run 5 seconds and then interrupt
   source.request_stop();
   thr1.join();
   thr2.join();
}
```

condition_variable can wait only on *unique:lock<mutex>*. *condition_variable_any* can wait on any lockable object (i.e. on anything that has *lock()* and *unlock()* methods). In addition, with *condition_variable_any* we can stop waiting not only with notification but also with interrupting the thread.

Multithreading (6)

```
void Producer(mutex* pmx, condition_variable_any* pcv, queue<int>* pvalues,
                    stop_token stop)
{
   default_random_engine generator;
   uniform_int_distribution<int> random_number(0, 100);
   unique_lock<mutex> lock(*pmx);
   for (int i = 0; i < 10; i++) {
      if (stop.stop_requested())   {
         cout << "Stop request detected" << endl;
         while (!pvalues->empty()) {
            pvalues->pop();
         }
         return;
      }
       pvalues->push(random_number(generator));
       this_thread::sleep_for(chrono::milliseconds(1000));
   }
   pcv->notify_one();
   cout << endl;
}
```

Multithreading (7)

```
void Consumer(mutex *pmx, condition_variable_any *pcv, queue<int> *pvalues,
                    stop_token stop) {
   unique_lock<mutex> lock(*pmx);
   pcv->wait(lock, stop, [&]() { return !pvalues->empty(); });
   if (stop.stop_requested()) {
      cout << "Waiting broken off" << endl;
      return;
   }
   while (!pvalues->empty()) {
      cout << pvalues->front() << ' ';
     pvalues->pop();
   }
   cout << endl;
}
```

Multithreading (8)

C++ v. 20 has some new tools for synchronization of threads. One of them is the *latch*:

```
#include <latch> // see https://en.cppreference.com/w/cpp/thread/latch
void Test ()  {
    vector<int> v1, v2;
    latch data_ready(2), clear_data(1);
    // Latch has a counter, its initial value is the parameter of constructor
    // There is no possibility to increase or reset the value of counter later
    jthread thr1(Producer1, &v1, 10, 500, &data_ready, &clear_data); // see the next slide
    jthread thr2(Producer2, &v2, 5, 200, &data_ready, &clear_data);
    data_ready.wait();  // Wait until the latch counter is zero. The threads decrement the
                        // counter. So, Test() can continue when the both vectors are filled
    ranges::sort(v1);
    ranges::sort(v2);
    vector<int> v(15);
    ranges::merge(v1, v2, v.begin());
    ranges::for_each(v, [](const int& i) { cout << i << ' ';  });
    cout << endl;
    clear_data.count_down(); // decrements the counter, allow the threads to continue
    thr1.join();
    thr2.join();
}
```

Multithreading (9)

```
void Producer1(vector<int> *pvec, int n, int t, latch *pdata_ready, latch *pclear_data) {
  default_random_engine generator;
  uniform_int_distribution<int> random_number(0, 100);
  for (int i = 0; i < n; i++) {  // fill the vector
    pvec->push_back(random_number(generator));
    this_thread::sleep_for(chrono::milliseconds(t));
  }
  pdata_ready->count_down();  // atomically decrements the latch counter
  pclear_data->wait();  // waits until latch clear_data decremented by Test() becomes 0
  pvec->clear();  // data is consumed, we can now delete it
}
void Producer2(vector<int>* pvec, int n, int t, latch *pdata_ready, latch *pclear_data) {
  default_random_engine generator;
  binomial_distribution<int> random_number(100);
  for (int i = 0; i < n; i++) {
    pvec->push_back(random_number(generator));
    this_thread::sleep_for(chrono::milliseconds(t));
  }
  pdata_ready->count_down();
  pclear_data->wait();
  pvec->clear();
}
```

Multithreading (10)

The barrier is more flexible:

```
#include <barrier> // see https://en.cppreference.com/w/cpp/thread/barrier
void Test ()
{
   vector<int> v1, v2;
   barrier<> data_ready(3);  // barrier for 3 tasks, template parameters by default
   jthread thr1(Producer1, &v1, 10, 500, &data_ready); // see the next slide
   jthread thr2(Producer2, &v2, 5, 200, &data_ready);
   data_ready.arrive_and_wait();
   // The first task is performed: the threads are launched
   // Waits until the 2 tasks implemented by Producer1 and Producer2 are performed
   // When all the 3 tasks are marked as done, stops waiting
   ranges::sort(v1);
   ranges::sort(v2);
   vector<int> v(15);
   ranges::merge(v1, v2, v.begin());
   ranges::for_each(v, [](const int& i) { cout << i << ' ';  });
   cout << endl;
   thr1.join();
   thr2.join();
}
```

Multithreading (11)

```
void Producer1(vector<int>* pvec, int n, int t, barrier<> *pdata_ready) {
  default_random_engine generator;
  uniform_int_distribution<int> random_number(0, 100);
  for (int i = 0; i < n; i++)    {
    pvec->push_back(random_number(generator));
    this_thread::sleep_for(chrono::milliseconds(t));
  }
  pdata_ready->arrive_and_wait();
          // marks the task as done (vector is filled), waits until all the tasks are closed
}
void Producer2(vector<int>* pvec, int n, int t, barrier<> *pdata_ready) {
  default_random_engine generator;
  binomial_distribution<int> random_number(100);
  for (int i = 0; i < n; i++) {
    pvec->push_back(random_number(generator));
    this_thread::sleep_for(chrono::milliseconds(t));
  }
  pdata_ready->arrive_and_wait();
}
```

Multithreading (12)

The barrier may have a callback function. It is invoked when all the tasks are marked as done. Normally it is implemented as a a functor:

```
class Msg {
 public: void operator() () noexcept { cout << "Ready" << endl; }  // noexcept is necessary
};
void Producer1(vector<int>* pvec, int n, int t, barrier<Msg> *pdata_ready) {
……………………
}
void Producer2(vector<int>* pvec, int n, int t, barrier<Msg> *pdata_ready) {
……………………
}
void Test ()  {
   barrier data_ready(3, Msg()); // template parameters are deduced from functor
……………………
}
```

When all the tasks are done and the *arrive_and_wait()* method returns, the number of tasks specified in the constructor is reset. Thus, a barrier may be used in loops. See the example on the following slides.

Multithreading (13)

```
void Test ()
{
  barrier<> data_ready(2);
  vector<int> v;
  default_random_engine generator;
  for (int i = 0; i < 5; i++)
  {
    jthread* pthr1 = new jthread { Producer, &generator, &v, 10, &data_ready };
    jthread* pthr2 = new jthread { Consumer, &v, &data_ready };
    pthr1->join();
    pthr2->join();
    delete pthr1;
    delete pthr2;
  }
}
```

Multithreading (14)

```
void Producer(default_random_engine *pgen, vector<int>* pvec, int n, barrier<>* pready)
{
  uniform_int_distribution<int> random_number(0, 100);
  pvec->clear();
  for (int i = 0; i < n; i++)
  {
    pvec->push_back(random_number(*pgen));
    this_thread::sleep_for(chrono::milliseconds(500));
  }
  pready->arrive_and_wait();
}

void Consumer(vector<int>* pvec, barrier<>* pready)
{
  pready->arrive_and_wait();
  ranges::for_each(*pvec, [&](const int& i)
      {
        cout << i << ' ';
        this_thread::sleep_for(chrono::milliseconds(500));
      });
  cout << endl;
}
```

Multithreading (15)

C++ v. 20 has two types of semaphores:

```
#include <semaphore> // https://en.cppreference.com/w/cpp/thread/counting_semaphore
int max_value, initial_value;
counting_semaphore<max_value> sem1(initial_value);
binary_semaphore sem2(initial_value);  // max_value is 1, initial_value may be 0 or 1
```

Method *release()* atomically increments the counter, method *acquire()* decrements it. The counter cannot be negative and cannot be greater than the *max_value*.
If the counter has become zero, *acquire()* blocks the thread. If due to call to *release()* the counter has a positive value, the blocked thread can continue.

An example about usage of semaphores: suppose we have a server that must process requests. For processing a new request we have to launch a new thread. However, the number of threads running concurrently cannot be endless.

To solve the problem we start the program with semaphore in which the *initial_value* is set to max. Each thread starts with call to *acquire(),* i.e. with starting the thread we decrement the counter. If the counter becomes zero, max allowed number of threads are already running and the new thread must wait. Each thread ends with call to *release()*, i.e. with ending the thread we increment the counter. If the counter was 0, it is now 1 and the thread that was blocked may start to run.

The *binary_semaphore* can replace mutexes. See the example on the next slide.

Multithreading (16)

```
void Producer(vector<int>* pvec, int n, binary_semaphore* pdone) {
    default_random_engine generator;
    uniform_int_distribution<int> random_number(0, 100);
    for (int i = 0; i < n; i++) {
        pvec->push_back(random_number(generator));
       this_thread::sleep_for(chrono::milliseconds(500));
    }
    pdone->release();
}
void Consumer(vector<int>* pvec, binary_semaphore* pdone)  {
    pdone->acquire();  // blocked until the Producer increments the counter
    ranges::for_each(*pvec, [&](const int& i)  { cout << i << ' '; });
    cout << endl;
}
void Test () {
    binary_semaphore done(0);  // initially in state 0
    vector<int> v;
    jthread thr1(Producer, &v, 10, &done);
    jthread thr2(Consumer, &v,&done);
    thr1.join();
    thr2.join();
}
```