SYCL 101

Contents

SYCL 101

Welcome to the **SYCL 101** course! In today's fast-paced world of computing, harnessing the full potential of hardware acceleration has become a necessity. SYCL, an open-standard, single-source programming model, empowers developers to unlock the incredible power of heterogeneous computing environments.

This comprehensive course is designed to provide you with a solid foundation in SYCL programming, equipping you with the knowledge and skills to leverage modern C++ for high-performance computing tasks. Whether you're a seasoned developer or just starting your journey in heterogeneous computing, SYCL 101 will guide you through the essential concepts and best practices to succeed in this dynamic field.

Course Outline:

Heterogeneous Compute: We begin by delving into the world of heterogeneous computing, understanding the diverse hardware accelerators available today, and how SYCL enables seamless integration.

Why SYCL?: Explore the compelling reasons why SYCL stands out as a powerful tool for GPU programming, FPGA development, and other heterogeneous compute environments. Basic real-world use cases will illustrate SYCL's versatility.

Modern C++: SYCL and modern C++ are a formidable combination. In this section, we'll cover essential C++ features and how they enhance SYCL programming, ensuring you're well-prepared for advanced topics.

SYCL Implementations of Modern C++: Dive into memory management with a comparison of conventional approaches and Unified Shared Memory (USM). Discover how USM can optimize your SYCL code and improve performance.

Next Step: SYCL Essentials: As you wrap up the course, we'll introduce the free SYCL Essentials course as your best next step in your SYCL journey to continue exploring this exciting field.

By the end of SYCL 101, you'll not only grasp the fundamental concepts of SYCL but also have basic practical skills to embark on more complex SYCL projects with confidence. So, let's get started on your journey to mastering heterogeneous computing with SYCL!

1) Heterogeneous Compute

Heterogeneous vs Homogeneous Computing Environments

Homogeneous Compute

Homogeneous computing refers to a system where all processing units are of the same type and have similar capabilities. In this setup, multiple identical processors work together to execute tasks in parallel. On the other hand, heterogeneous computing involves a system that combines different types of processing units, such as Central Processing Units (CPUs), Graphics Processing Units (GPUs), Field-Programmable Gate Arrays (FPGAs), or specialized accelerators. Each processing unit is designed to perform specific tasks efficiently.

One may choose homogeneous computing when the workload is well-suited for parallel execution on multiple identical processors. It simplifies programming as the same code can be executed on all processors, and load balancing among processors is relatively straightforward. Homogeneous computing is often employed in high-performance computing (HPC) clusters and distributed systems that require large-scale parallel processing.

Heterogeneous Compute

Heterogeneous computing is preferred when the workload consists of tasks that can benefit from different types of processing units. For example, GPUs excel at parallelizing graphics and general-purpose computing tasks, while FPGAs offer highly customizable and efficient processing for specific algorithms. By utilizing specialized accelerators, heterogeneous computing systems can achieve higher performance and energy efficiency for certain workloads.

Advantages of Accelerators in Heterogeneous Computing

1. Enhanced performance: Accelerators are specifically designed to excel at certain types of computations. By offloading specific tasks to accelerators, overall performance can be significantly improved.
2. Energy efficiency: Accelerators are often optimized for a particular type of algorithm, allowing for better performance per watt compared to general-purpose processors.
3. Ability to further optimize code: Developers can use accelerators to tailor the hardware and software for specific algorithms or applications, enabling higher performance and efficiency.

Challenges using Accelerators in a Heterogeneous Computing Environment

1. Programming complexity: Unlike homogeneous systems, heterogeneous computing requires different programming models and languages for various processing units. This complexity increases the development effort.
2. Data movement: Efficiently moving data between different processing units can be a bottleneck. Developers must carefully manage data transfers between host and device memory in order to minimize overhead.
3. Load balancing: In a heterogeneous system, workload distribution and load balancing become more complex due to the diverse capabilities of different processing units. Proper load balancing is essential to fully utilize the system's resources.

Software Development Techniques

The software development techniques for homogeneous and heterogeneous computing have typically required different approaches in the past.

In homogeneous computing, the programming model was typically based on parallel processing paradigms such as message passing (MPI) or multithreading (OpenMP). Developers focused on parallelizing code across multiple identical processors and optimizing data dependencies and synchronization.

Heterogeneous computing typically used different programming models and APIs for each type of accelerator and CPU. For example, CUDA and ROCm are commonly used for programming vendor-specific GPUs, requiring developers to have a unique tool chain for each development target. In addition, developers would need to become familiar with how to design and optimize algorithms for each type of accelerator as well as how to best partition the workload and manage data movement between different hosts and devices.

Summary

Homogeneous computing utilizes identical processors for parallel processing, while heterogeneous computing combines different processing units to achieve performance and efficiency gains for specific tasks. The choice between them depends on the nature of the workload. Accelerators in heterogeneous computing offer advantages in performance and efficiency but introduce challenges related to programming complexity, data movement, and load balancing. Software development techniques also differ, with homogeneous computing focusing on parallelization across identical processors and heterogeneous computing requiring different programming models for each processing unit.

The oneAPI specification was created to address the challenges of managing multiple tool chains and multiple development environments. It delivers a unified software development environment across CPU and accelerator architectures.

Khronos SYCL

The SYCL programming language from the Khronos Group offers several advantages for targeting heterogeneous compute environments. Here are some key benefits:

Open Ecosystem

SYCL is part of the broader oneAPI initiative, which aims to provide a unified and open programming model for heterogeneous computing. It is developed by an industry consortium, ensuring broad industry support and collaboration. The open ecosystem approach promotes interoperability and compatibility across different hardware vendors, enabling developers to write code that can run on various accelerators and processors.

Performance Across Devices

SYCL allows developers to write code that can be executed efficiently across different devices, including CPUs, GPUs, FPGAs, and other accelerators. SYCL leverages the power of modern hardware architectures by expressing parallelism and data dependencies through high-level abstractions, such as kernel functions and data accessors. This enables performance portability, allowing code to adapt and scale across diverse compute devices while taking advantage of their specific capabilities.

Ease of Coding

SYCL is a familiar single-source, event-based programming model based on standard C++. Developers can leverage their existing C++ knowledge and skills to write code for heterogeneous systems. SYCL provides a higher-level abstraction than lower-level APIs like CUDA or OpenCL, making it easier to express parallelism and offload computations to accelerators. The SYCL programming model includes features such as task and data parallelism, memory management, and synchronization mechanisms, simplifying the development process for heterogeneous computing.

Portability and Future-Proofing

SYCL promotes code portability across different hardware architectures. Developers can write SYCL code to target multiple devices with minimal modifications, reducing the effort required to adapt their applications to different hardware platforms. Additionally, the open ecosystem and industry collaboration behind oneAPI provides a level of future-proofing, ensuring that SYCL remains compatible with new generations of hardware and software advancements.

Integration with Existing Codebases

SYCL allows developers to integrate accelerators and heterogeneous computing capabilities into their existing software projects. By leveraging SYCL, developers can selectively offload computationally intensive tasks to accelerators while keeping the rest of their codebase unchanged. This integration capability enables incremental adoption of heterogeneous computing and maximizes the benefits of existing software investments.

Summary

Overall, SYCL offers the advantages of an open ecosystem, performance portability across devices, and ease of coding by providing a high-level, C++-based programming model. It enables developers to efficiently target heterogeneous compute environments, leverage diverse hardware capabilities, and simplify the development process for accelerated computing applications.

2) Why SYCL?

What is SYCL?

Definition

SYCL is a high-level single-source standard C++ programming model that can target a wide range of heterogeneous platforms. SYCL is an industry-driven standard developed by Khronos Group and announced in March 2014 that allows data parallelism for heterogeneous systems based on C++ 17. It is a royalty-free, cross-platform abstraction layer that enables the development of applications and frameworks with the portability and efficiency of OpenCL, while providing a user interface that's flexible and easy to use. SYCL standard is designed to be both highly portable and highly efficient and allows applications to offload compute kernels and data transfer to OpenCL-enabled accelerators such as GPUs, FPGAs, and other processors.

SYCL Features

- **SYCL is modern standard C++:** Unlike many other programming models and languages that require language extensions, SYCL does not need any of that. It is pure standard C++; any SYCL application will compile and run using a SYCL library, even if a device compiler and a device are not available in your system.
- **SYCL is open, multivendor, and multiarchitecture:** Because there is no need to copy data, it enables pointer-based algorithms and data structures to be used across architectures of different vendors and makes it easier to use the same code on different devices.

- **Separation of data access and data storage:** When using buffers and accessors in SYCL, the data access and data storage are kept separate. This allows the programmer to create data-parallel tasks easily, without needing to manually track data movement and event dependencies between kernels.
- **No need to copy data:** Unified Shared Memory (USM) allows different devices, such as CPUs and GPUs, to share and access data without having to copy it between them.
- **Ease of hierarchical parallelism:** Inspired by OpenCL and OpenMP, SYCL makes data parallelism easy to understand by using hierarchical parallelism and modern C++ instructions, so tasks are easily organized and synchronized, which makes the code more efficient and less fragmented.
- **Two compilation phases:** SYCL requires two compilation passes, one for the host code and one for the device code. However, unlike OpenCL, SYCL is single-source, meaning there is only one executable. The two compilations can be made by the same compiler.
- **High-level programming model:** SYCL provides several high-level abstractions over common boilerplate code in OpenCL and other backends. This includes selecting devices, allocating storage, copying data, managing dependencies, scheduling kernels, and much more.

Specifications

SYCL utilizes specifications to standardize and streamline the development of parallel applications for heterogeneous architectures, enabling portability, productivity, and performance while fostering a vibrant community of SYCL users and providers. New specifications are being released to accommodate new features and technological advances.

The SYCL specifications that have been published as of today are:

- **SYCL 1.2**: Specification released in May of 2015: It is considered obsolete.
- **SYCL 2.2**: It was a provisional specification published in 2016 that is considered deprecated.
- **SYCL 1.2.1**: Its latest revision was released on April 27, 2020.
- **SYCL 2020**: Specification revision 7 released on April 18, 2023.

The SYCL 2020 specification marks a significant leap forward that features over 40 new improvements, such as:

- **Unified Shared Memory (USM)**, enabling code with pointers to work naturally without buffers or accessors.
- **Parallel reductions**, adding a built-in reduction operation and helping to avoid boilerplate code, providing maximum performance for hardware with built-in operations.
- **Work group and sub-group algorithms**, enabling efficient operations between work items.
- **Class template argument deduction (CTAD)** and deduction guides to enable simpler class template instantiation.
- **Simplification of accessors**, which adds a built-in reduction operation, reduces the burden of boilerplate code, and enables simplified C++ patterns.
- **Expanded interoperability** with different backends, enabling support for backends other than OpenCL.
- **Improvements to atomic operations** to be closer to C++ atomics to enable more parallel programming freedom.

Implementations

A SYCL implementation refers to a software version or revision of the SYCL programming model. It consists of the necessary tools, libraries, and runtime support that enable developers to write SYCL code and execute it on compatible hardware platforms. Different vendors and organizations can provide their own SYCL implementations, each with its own set of features, optimizations, and supported hardware platforms. These implementations provide the necessary compiler, libraries, and runtime components to enable developers to write and execute SYCL code efficiently on their respective platforms.

These are some of the popular SYCL implementations that are available:

- **DPCP++** by Intel as part of the oneAPI solution that supports Intel CPUs, GPUs, and FPGAs as well as NVIDIA and AMD GPUs

- **ComputeCpp** by Codeplay
- **TriSYCL** by Xilinx
- **Open SYCL** (formerly hipSYCL) by the University of Heidelberg
- **neoSYCL** by the Tohoku University
- Source: Khronos (https://www.khronos.org/sycl/)

SYCL Basic Code: single_task

SYCL offers many advantages to developers, including improved productivity, better performance, and enhanced code maintainability. In this lesson, you will gain an overall understanding of basic SYCL features and concepts.

SYCL Advantages

1. SYCL provides an **easier programming model for heterogeneous applications** than traditional models such as OpenCL or CUDA.

2. SYCL **supports modern C++ language features** and can help simplify writing portable and maintainable code.

3. SYCL allows developers to take advantage of heterogeneous hardware architectures and **utilize multiple processors or accelerators simultaneously**.

4. SYCL provides an abstraction layer that makes it **easier to port code to different hardware** architectures.

5. SYCL enables the development of **high-performance** and **data-parallel** applications.

6. SYCL **allows third-party vendors to provide tools** that help optimize code for different hardware architectures.

Basic SYCL Code #1: single_task

In this subsection, we present a code with a simple SYCL algorithm called `single_task`, which equivalent in standard C++ would be `std::thread`, to gain a fundamental understanding of SYCL components. **The construct**`single_task`**is used to define a unit of work that should be executed on a single processing element, typically a single CPU core or GPU thread.** Our goal in this code is to modify specifically the 6th element of a 10-size vector using the SYCL `single_task` command. We will provide a comprehensive explanation of each segment of the code.

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {

    queue Q;                          // The queue, Q, is the object that
                                      // submits the task to a device.
    int const size = 10;
    buffer<int> A{ size };            // The buffer, A, is the memory used to
                                      // transfer data between host and device.

    Q.submit([&](handler& h) {        // The handler, h, is the object that contains
                                      // the single_task function to be used.

        accessor A_acc(A, h);         // The accessor, A_acc, is the object that
                                      // efficiently accesses the buffer elements.

        h.single_task([=]() {
            A_acc[5] = 77;
```

```
            });

        });

    host_accessor result(A);        // host_accessor is the object that allows
                                    // the host to access the buffer memory.

    for (int i = 0; i < size; i++)  // Print output
        std::cout << result[i] << " "; std::cout << "\n";
    return 0;

}
```

In SYCL code, several fundamental components play vital roles in coordinating work, managing data, and ensuring seamless communication between the host and device. One of these essential elements is the `queue`, denoted as `Q`. The queue serves as the central construct used to submit work items, control execution flow, and facilitate data transfers in a parallel and heterogeneous computing environment, such as CPUs or GPUs.

Another critical component is the `buffer`, represented as `A`. This `buffer` serves as a data container that defines the region of memory accessible by both the host and the device, enabling efficient data sharing and transfer between them.

Another fundamental concept in SYCL programming is the **command group** that, in our code, it is defined by:

```
Q.submit([&](handler& h) {

    ...

}
```

The **command group** represents a unit of work that can be submitted to a SYCL queue for execution; its main function is to define the operations or computations that are to be performed on the target device.

The command group needs the `accessor`:

```
accessor A_acc(A, h);
```

The `accessor`, `A_acc`, is the object used to define the access rights (`read-only`, `write-only`, or `read-write`) of specific kernels to the buffer elements.

Inside of the command group resides the specific **SYCL kernel function**, which is `single_task` in this case. Note that `single_task` is provided by the handler, `h`:

```
h.single_task([=]() {
    A_acc[5] = 77;
});
```

The `handler` is the object that represents a context in which command groups are defined. It specifies the operations and dependencies within a command group and controls the execution behavior of those operations. One consideration to keep in mind is that only one SYCL **kernel function**, even if it is the same, can be executed in the command group.

The **kernel function** in this case is:

```
A_acc[5] = 77;
```

Note that a lambda function can be used as kernel function as well.

Basic SYCL Code #2: parallel_for Hands-on

To become familiar with the SYCL structure, we propose the following hands-on exercise for you to think about:

Create a variation of the`single_task`code that changes, in one step, all elements of a 10-size vector to the value of 77.

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {

    queue Q;                          // The queue, Q, is the object that
                                      // submits the task to a device.
    int const size = 10;
    buffer<int> A{ size };            // The buffer, A, is the memory used to
                                      // transfer data between host and device.



    //********** YOUR CODE STARTS HERE **********//

    // Step 1) Submit the queue with the handler definition:

    // Step 2) Define the accessor with buffer and handler:

    // Step 3) Call parallel_for from the handler specifying range and index:

    // Step 4) Change the accessor elements with the desired value using the index:


    //********** YOUR CODE ENDS HERE **********//



    host_accessor result(A);          // host_accessor is the object that allows
                                      // the host to access the buffer memory.

    for (int i = 0; i < size; i++)  // Print output
        std::cout << result[i] << " "; std::cout << "\n";
    return 0;

}
```

If you are thinking that `single_task` might not be the best approach you are in the right track.

> **NOTE** The best way to tackle this is with the use of `parallel_for`.

The solution to this `parallel_for` hands-on exercise is in the next section.

Main SYCL concepts takeaways

To recap, let's revisit the main concepts introduced in this section for expressing parallelism, managing data transfers, and controlling the execution of workloads in SYCL programs to enable efficient execution on heterogeneous platforms:

- **Queue**: A SYCL queue manages the execution of command groups on a specific device. It acts as a command queue, allowing you to submit command groups for execution and control the order of execution.
- **Scheduler**: The scheduler in SYCL is an internal component of the runtime system that manages the execution and scheduling of command groups on devices. It optimizes the execution by considering device capabilities, workload distribution, dependencies, and resource availability. The queue is the primary interface through which tasks are submitted to the scheduler for execution.
- **Buffer**: A buffer in SYCL is a data container that represents a region of memory accessible by both the host and the device. It enables efficient data transfer and sharing between the host and the device without explicit memory management.
- **Accessor**: Data represented by a buffer cannot be directly accessed through the buffer object. Instead, we must create accessor objects that allow us to safely access a buffer's data. Accessors inform the runtime where and how we want to access data, allowing the runtime to ensure that the right data is in the right place at the right time.
- **Command group**: In SYCL, a command group represents a unit of work that is submitted for execution on an OpenCL device. It encapsulates a set of operations and allows you to express parallelism and dependencies between tasks. Remember to call only a SYCL kernel function per command group.
- **Handler**: A handler in SYCL represents a context in which command groups are defined. It provides methods for specifying operations within a command group, such as kernel invocations and memory transfers, and controls the execution behavior of those operations.

SYCL Basic Code: parallel_for

In this section, we will explain `parallel_for`, which is a core concept in SYCL designed to facilitate parallel execution of code across a range of processing elements, such as GPU threads or CPU cores. **Essentially**`parallel_for`**distributes work across multiple processing elements for parallel execution and allows developers to express parallelism easily .** The equivalent of `parallel_for` in standard C++ would be `std::for_each`. With `parallel_for`, programmers only have to define a function or a lambda expression that represents the work to be done in parallel while maintaining code portability.

Here's an example of how `parallel_for` works (solution of previous section):

```
#include <CL/sycl.hpp>
using namespace sycl;

int main() {

    queue Q;                          // The queue, Q, is the object that
                                      // submits the task to a device.
    int const size = 10;
    buffer<int> A{ size };            // The buffer, A, is the memory used to
                                      // transfer data between host and device.

    Q.submit([&](handler& h) {        // The handler, h, is the object that contains
                                      // the parallel_for function to be used.

        accessor A_acc(A, h);         // The accessor, A_acc, is the object that
                                      // efficiently accesses the buffer elements.

        h.parallel_for(range<1>(size), [=](id<1> indx) {
            A_acc[indx] = 77;
            });

        });

    host_accessor result(A);          // host_accessor is the object that allows
                                      // the host to access the buffer memory.
```

```
    for (int i = 0; i < size; i++)  // Print output
        std::cout << result[i] << " "; std::cout << "\n";
    return 0;

}
```

Note that

```
h.parallel_for(range<1>(size), [=](id<1> indx) {
    A_acc[indx] = 77;
    });
```

is where `parallel_for` is presented within the code and, as it happened with `single_task`, `parallel_for` is also provided by the handler `h`. Let's explain these components:

- `h.parallel_for`: This line initiates a parallel computation using the parallel_for construct. It means that the enclosed code block will be executed in parallel across multiple processing elements (e.g., CPU cores or GPU threads). The `h` is the handler, which manages the execution of SYCL tasks on a specific device.
- `range<1>(size)`: This part specifies the range of work items for the parallel computation. In this case, it's a 1D range defined by `range<1>` with a size of `size=10`. The range defines how many times the enclosed code block will execute in parallel meaning that in this case the code block will be executed 10 times concurrently, each time with a different value of `indx` ranging from 0 to 9.
- `[=](id<1> indx) { A_acc[indx] = 77; }`: This is a lambda function that represents the work to be performed in parallel. The lambda function takes an `id<1>` argument named `indx`, which represents the unique identifier of the current work item. Inside the lambda function, the code sets the value at the indx-th position of the array `A_acc` to `77`. In essence, each parallel instance will update a different element of the `A_acc` array to the value 77.

Note that it is not the buffer, `A`, that is modified inside the SYCL kernel function, but the accessor, `A_acc`, which manages the access and usage of the buffer memory.

Conclusions

After witnessing the remarkable ease and convenience that SYCL offers through its `single_task` and `parallel_for` examples, it becomes evident why SYCL is an excellent choice for harnessing the power of heterogeneous computing. **SYCL's user-friendly approach**, as demonstrated by these constructs, **accelerates development, improves code readability, and simplifies parallel programming complexities.** Beyond this, SYCL's inherent cross-platform portability **ensures that your code can effortlessly adapt to diverse hardware architectures**, eliminating the need for extensive platform-specific optimizations.

Remarkably, **SYCL achieves this while preserving performance**, allowing you to tap into the full potential of heterogeneous hardware with ease. Furthermore, SYCL's seamless integration with modern C++ and its robust ecosystem support make it a pragmatic choice for developers seeking to future-proof their applications in a rapidly evolving technological landscape. **In essence, SYCL combines convenience, performance, and versatility, making it an ideal tool for those striving to create efficient and portable code for a wide range of computing platforms.**

The benefits of using SYCL are multiple and diverse. However, before we delve deeper into SYCL's capabilities, it is first necessary to cover some of the most powerful C++ features. This is addressed in the following section: **Modern C++** of this SYCL 101 course.

3) Modern C++

- Uniform Initialization
  - Introduction

Uniform Initialization

This chapter covers uniform initialization. You will learn the following:

1. What is uniform initialization and how can it be used?
2. Why should uniform initialization be used?
3. What are the common problems with uniform initialization?

Introduction

In modern C++, there is a uniform method for initializing data called **uniform initialization**.

Expression Initialization

To better understand the concept, let's get familiar with the following terms:

- **direct initialization**, which uses an explicit set of constructor arguments to create an object.
- **copy initialization**, which uses another object to initialize an object.

The code below shows both direct and copy initialization:

```
std::string direct("direct initialization");
std::string copy = "copy initialization";
```

Brace Initialization

To uniformly initialize objects of any type, **brace initialization form {}** may be used for both direct and copy initialization. When used with brace initialization, we call initializations direct-list and copy-list.

The code below shows both direct-list and copy-list initialization:

```
std::string direct{"direct-list initialization"};
std::string copy = {"copy-list initialization"};
```

Let's look at uniform initialization on different built-in and custom types:

1. Built-in types:

```
int i {13};
float f {2.7};
```

2. Arrays:

```
int my_array[5] {0, 1, 2, 3, 4};
```

3. Dynamically allocated arrays:

```
int* my_array = new int[5]{0, 1, 2, 3, 4};
```

4. Standard library containers:

```
std::vector<int> my_vector{0, 1, 2, 3, 4};
std::map<int, std::string> my_map{{1, "str1"}, {7, "str2"}};
```

5. User-defined types:

```
class foo{
   public:
      foo() : _i(0), _f(0.0) {}
      foo(int i, float f) : _i(i), _f(f) {}

   private:
      int _i;
      float _f;
};

foo f1{};
foo f2{13, 2.7};
```

Why Should Uniform Initialization Be Used?

We can list several advantages of uniform initialization.

Consistent Syntax

The first is **very consistent syntax**. For example, we already know there are many ways to initialize the variable:

```
int i = 1;   // historically the most common way
int i(1);    // direct initialization
int i{1};    // direct-list initialization
int i = {1}; // copy-list initialization
auto i{1};   // direct initialization of type deduced to int
```

For simple type initialization, it's not a problem to use the most common method. However, when we use different, more complicated custom types, the consistent syntax can change the experience with code. This can be especially important if you consider the generic code that should be able to initialize any type — it will be not possible with `()` initialization.

```
int i{1};
foo f{13, 2.7};
std::vector<int> v{0, 1, 2, 3, 4};
std::unordered_set<int> s{13, 17, 8};
std::unordered_map<int, std::string> {{1, "one"}, {2, "two"}};
```

Narrowing Conversions Are Not Allowed

The second benefit is that uniform initialization **does not allow narrowing conversions**.

Before uniform initialization, with C-style C++, the code below will work, and `double` will just convert to `int`.

```
double d = 5.5;
int i = d; // double to int conversion
```

The same with bracket initialization will not work and it forces the user to type-cast values explicitly.

```
int i{d}; // compilation error

int i{static_cast<int>(d)}; // modern C++ cast - best practice
int i{(int)d};              // C-style type-cast
int i{int(d)};              // old C++-style type-cast
```

Fixes Most Vexing Parse

The most vexing parse comes from a rule in C++ that says that anything that could be considered a function declaration should be parsed by the compiler as a function declaration.

Let's examine the example when we want to initialize the vector that's a private member of the foo class with three zeros `{0, 0, 0}`,

```
class foo{
   public:
      foo() { ... }

   private:
      std::vector<int> v(3, 0);
};
```

This code will not compile because the vector initialization was interpreted by the compiler as a function declaration. We have three possible solutions for this problem.

The first is the most obvious — we can just use uniform initialization for the vector.

```
std::vector<int> v{0, 0, 0};
```

This is not always the best solution, especially when we need to initialize the long vector and typing every element is not an option.

The second solution is to move the initialization to the constructor:

```
foo() : v(3, 0) { ... }
```

And the last solution is to use copy initialization:

```
std::vector<int> v = std::vector<int>(3, 0);
```

Common Problems with Uniform Initialization

Even when uniform initialization helps, there are also some issues related to using it. The first of them is about using `auto` for variable declaration. Deduced type for the variable can be `std::initializer_list` instead of the type a programmer would expect. This happens mostly when we combine auto variable declaration with an equal sign or if it has multiple elements, as in the following code:

```
auto variable{13};       // variable is type of int
auto variable = {13};    // variable is of type std::initializer_list<int>

auto variable{13, 17, 8};    // compilation error variable contains multiple expressions
auto variable = {13, 17, 8}; // variable is of type std::initializer_list<int>
```

Another problem can happen with the vector initialization. It can be tricky, especially when a programmer is just learning C++. See the difference between declarations below:

```
std::vector<int> v(3,0); // vector contains tree zeros {0, 0, 0}
std::vector<int> v{3,0}; // vector contains three and zero {3, 0}
```

The last problem can be called "strongly prefer `std::initializer_list` constructors." It means that when calling the constructor using the uniform initialization syntax, the constructor will overload while declaring its parameter of type `std::initializer_list` (when it exists).

The example below demonstrates this situation:

```
class foo {
   public:
      foo(int i, float f) { ... }
      foo(std::initializer_list<bool> list) { ... }
};

foo object{13, 2.7}; // compilation error
```

The error occurs because instead of using the first constructor (with `int` and `float`), there is the constructor overload to the "strongly preferred" one with `std::initializer_list` as a parameter. So, the problem is caused by narrowing conversions from `int` and `double` to `bool`.

Multithreading

This chapter covers multithreading. You will learn the following:

1. What is multithreading?
2. What are the benefits of multithreading?
3. How multithreading is implemented in C++?
4. How to launch, join, and detach threads?
5. What are the problems with multithreading?

Introduction

Multithreading is a feature that allows concurrent execution of two or more parts of a program for maximum utilization of the resources. Each part of such a program is called thread. So, threads are lightweight sub-processes within a bigger process. In C++, multithreading was introduced in C++11, but became a part of the standard library (STL) in C++17.

Why Use Multithreading?

With the introduction of multiprocessor and multicore hardware, the use of multithreading started to be very important in terms of application efficiency.

There are various reasons to use multithreading:

- Higher throughput
- Better responsiveness
- Resource efficiency
- Parallelism

It is important to remember that even when using more and more threads, due to potential overhead the application may not always run faster.

Basic Concepts of Multithreading

Let's start with some basics concepts, like process and thread.

Process is what executes the program. Each process is able to run concurrent subtasks called threads.

Thread, as it was already explained, is a sub-task of the process. It can give the illusion that the application is performing multiple things all at once. Without threads, there will be a need to write one program per task and synchronize them at the operating system level.

Concurrency and Parallelism

Before we delve deeper into the topic of multithreading in C++, let's start by explaining the terms **concurrency** and **parallelism**. They are often used interchangeably, but it's important to understand their differences.

Put simply, **concurrency** is about multiple tasks which start, run, and complete in overlapping time periods, in no specific order, while parallelism is about multiple tasks or subtasks of the same task that run at the same time on hardware with multiple computing resources. It is important to understand that both parallelism and concurrency can occur separately as well as together depending on the context.

C++ Multithreading Syntax

Now, let's get back to multithreading itself. C++ multithreading involves creating and using thread objects, seen as `std::thread` in code, to carry out delegated sub-tasks independently.

Creating a Thread

Creating and launching a thread is really simple, e.g.:

```
#include <iostream>
// 1. We need to add the threads header to work with threads in C++
#include <threads>

// 2. Create a function that will be mapped to a thread
void callFromThread() {
   std::cout << "Hello world!\n";
}

int main() {
```

```
   // 3. Initialize thread and execute
   std::thread my_thread(callFromThread);

   // 4. Rejoin thread to the main thread
   my_thread.join();

   return 0;
}
```

Let's analyze this code step-by-step:

1. The first step imported the necessary library from the STL — it contains all the classes and functions related to the C++ multithreading like `std::thread` class.
2. The next step declared a function that was mapped to the thread — all threads must be given a function to complete at their creation.
3. The next step initialized a thread to execute the function — we are using default executor.
4. At the end, we used `join()` multithreading command — this task pauses the main function's thread until the specified thread completes. Without `join()`, the main thread could finish its task and terminate the process before the thread executing completes `callFromThread`. This race condition could result in an error.

Creating Multiple Threads

The previous example created a single thread (in addition to the main thread). We can just as easily create and execute multiple threads, e.g.:

```
#include <iostream>
// 1. We need to add threads header to work with threads in C++
#include <thread>
#include <vector>

// 2. Create a function that will be mapped to a thread
void print(int n, const std::string &str)  {
  std::string msg = std::to_string(n) + " : " + str + '\n';
  std::cout << msg;
}

int main() {
  std::vector<std::string> s = {
      "SYCL 101",
      "Intel",
      "multithreading",
      "education"
  };

  // 3. Initialize threads and execute them
  std::vector<std::thread> threads;
  for (int i = 0; i < s.size(); i++) {
    threads.push_back(std::thread(print, i, s[i]));
  }

  // 4. Rejoin threads to the main thread
  for (auto &th : threads) {
    th.join();
  }

  return 0;
}
```

This code is similar to the previous one-thread example:

1. First, we imported the `thread` library.
2. Then, we created a function that was mapped to the threads. In this example, the function is printing a given string and number.
3. Then, we initialized the threads and executed them. We created the `std::vector<std::threads>` to store the thread handles.
4. The last step rejoined the threads to the main thread.

In this case, as we are using multiple threads, it is important to mention that even though we initialized the threads in sequential order, there is no guarantee that they will execute in that order. You might see different output every time you run this program.

Joining and Detaching Threads

We've already used `join()` on the threads, but let's take a deeper look at join and detach operations. Joining threads is a form of synchronization that makes them wait for each other. Imagine that a thread is started, then another thread waits for this new thread to finish. In that scenario, we are calling the `join()` function on the `std::thread` object, like in the example below:

```
std::thread th(functionPointer);

// ...

th.join(); // waiting for the thread th to finish
```

In addition to **joining** threads, one can also **detach** them. A detached thread will continue without blocking or synchronizing its execution with any other threads. For this, we call `detach()` on the `ste::thread` object:

```
std::thread th(functionPointer);

th.detach(); // continue without waiting for thread th to finish
```

Remember that after calling `detach()`, `std::thread` object is no longer join with other threads in the process.

Commonly Used Methods of the Thread Class

We have introduced the class `std::thread` with its `join()` method, but `std::thread` has more. These are brief descriptions of the most relevant methods:

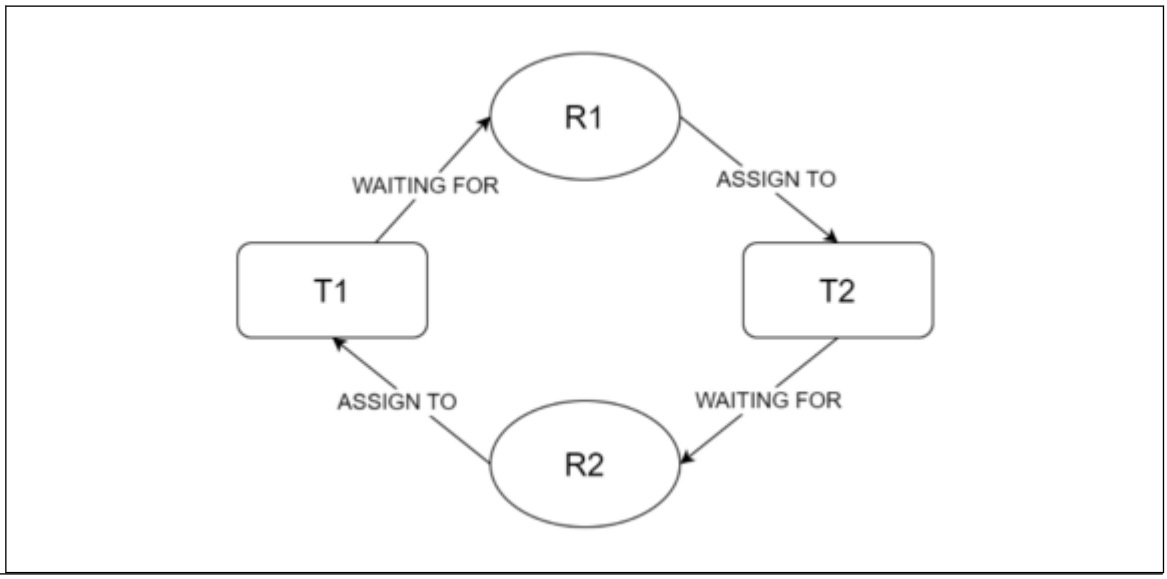
- `get_id()`: This returns a unique numerical identifier for the calling thread. A key application of this identifier is the facilitation of synchronization and thread-local storage, which is often used to managing static or global data that needs to be distinct for each individual thread.
- `interrupt()`: This method compels the thread halt immediately. The scheduler will ignore this thread, even if it is in the middle of a task. We recommend caution when using this method.
- `yield()`: This method informs the scheduler that the current thread is temporarily yielding control and can be revisited later. In a preemptive scheduling context, this is valuable to ensure that threads with lower priority tasks do not monopolize execution that could be more effectively utilized by other productive threads.
- `join()`: This suspends the execution of the current thread until the thread being joined completes its execution. It serves as the primary mechanism for thread synchronization. A typical scenario for its application involves the main thread initiating a background task within a separate thread, performing other operations in the meantime, and then pausing to ensure that the background task has concluded before proceeding further.

Problems with Multithreading

When running multithreaded programs we can face problems with **access to shared data** by multiple threads. Simultaneous access to the same resource can lead to race conditions, errors, and chaos in programs. This problem occurs mostly due to the consequences of modifying shared data. There will be no issue if the data we share is read-only because the data read by one thread is unaffected by whether or not another thread is reading the same data. However, once data is shared between threads and one or more threads begin modifying the data, difficulties arise. We will take a look at some different possible problems with shared data that can happen in multithreading programming.

Deadlock

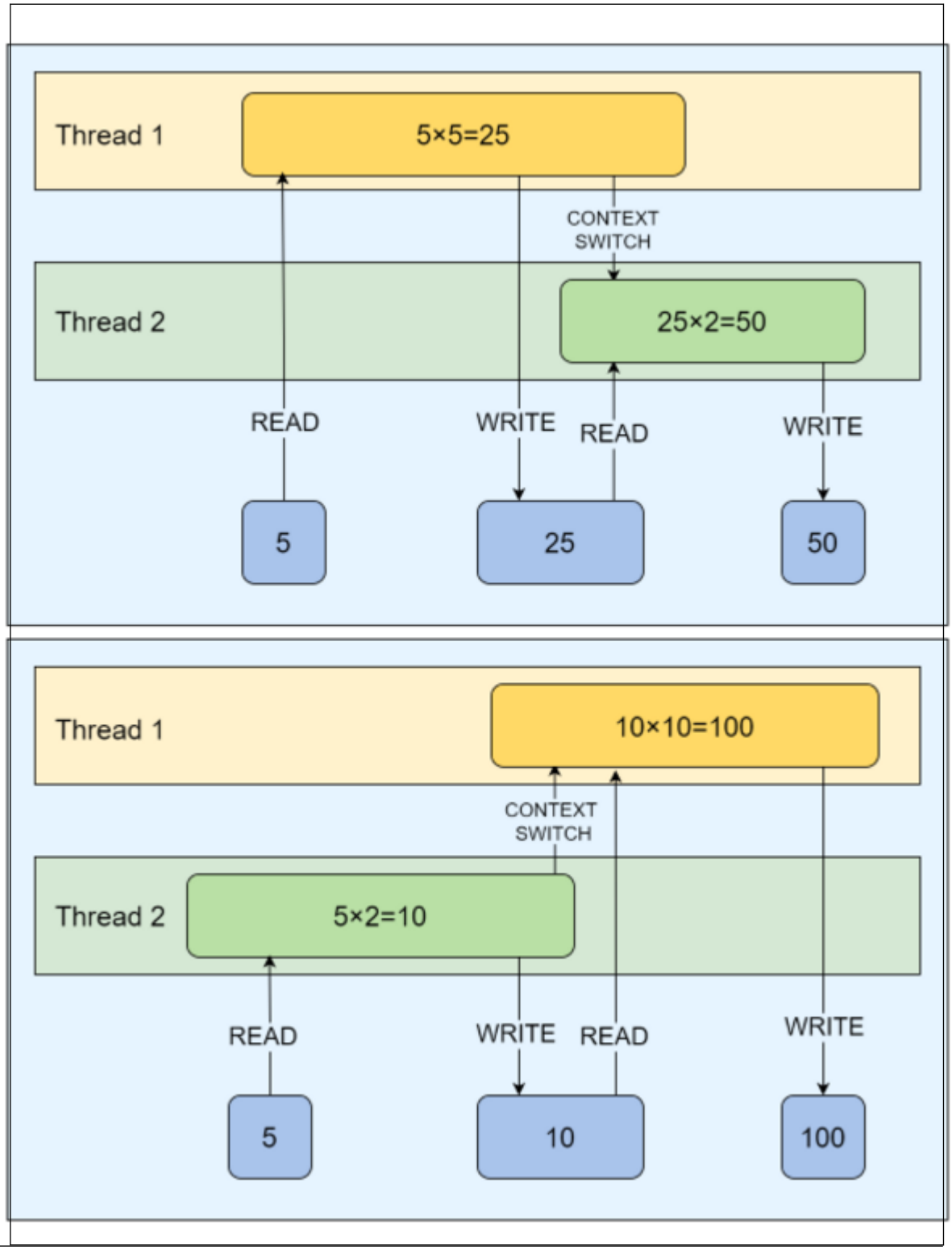
Deadlock is a situation where a thread cannot proceed because it is waiting for a resource that will never become available. Imagine the situation where we have two threads (T1 and T2) and two resources (R1 and R2). Thread T1 requires resource R1, and thread T2 requires resource R2. In that situation deadlock can arise when T1 is holding on R2 and waiting for R1 while at the same time, thread T2 is holding R1 and waiting for R2. This situation is depicted in the image below, which illustrates a bad locking cycle.



To avoid such deadlocks, shared resources should be acquired and released in reverse order. For example, a thread cannot acquire R2 unless it already holds R1, and it cannot release R1 until it first releases R2.

Race Conditions

A race condition occurs when threads can modify a shared resource in indeterminate order. This can produce incorrect results when correct results depend on a particular execution order. Imagine two threads doing different operations. The first takes a value and overwrites it with its square while the second takes the value and overwrites its double. Depending on the order of thread execution, the final value will be different:



As you can see in the first scenario, Thread 1 executed first so its result was doubled, which resulted in a final value of 50. In the second scenario, Thread 2 executed first so its resulted was squared, which resulted in a final value of 100.

To avoid race conditions, any operation that modifies a shared resource must be synchronized.

Summary

To summarize, multithreading is used to express concurrency in an algorithm and to execute independent tasks in parallel. It can increase the efficiency of a program but can also be tricky when dealing with shared resources. This was a very short introduction to multithreading. There is still much more to be learned.

Smart Pointers

This chapter covers smart pointers. You will learn the following:

1. What are smart pointers?
2. When should different smart pointer types be used?
3. Why should you use smart pointers?

What are Smart Pointers and When Should They be Used?

Let's start with the definition of smart pointers. Smart pointers are a type with values that may be used like pointers but with the added benefit of automated memory management. We have three types of smart pointers declared in the `<memory>` STD library:

- std::unique_ptr
- std::shared_ptr
- std::weak_ptr

In general, smart pointers are used in code that involves tracking the ownership of a piece of memory and allocating or deallocating it. They typically eliminate the necessity to do these things explicitly.

It is worth mentioning that regular pointers can be still used in code with oblivious memory ownership. This would typically be in functions that get a pointer from someplace else and do not allocate or deallocate memory and do not store a copy of the pointer that outlasts their execution. Let's take a deeper look at each of the smart pointer types.

std::unique_ptr

According to the C++ Reference:

`std::unique_ptr` is a smart pointer that owns and manages another object through a pointer and disposes of that object when the unique_ptr goes out of scope.

In other words, it is a smart pointer with **unique object ownership** semantics. It is a 1-to-1 relationship between a pointer and its allocated object on the heap. It's important to know that if the unique pointer is destructed, the allocated object on the heap is also destroyed.

Syntax

The unique pointer is declared as follows:

```
std::unique_ptr<int> ptr(new int); // allocation of new int on heap
```

As mentioned previously, the smart pointer has automated memory management. It will be destroyed at the end of the code block in which it was declared. Remember, the object it points to will also be destroyed.

```
{
  std::unique_ptr<int> ptr(new int);

  // pointer usage

} // ptr is destroyed, which means the int object is also destroyed
```

Usage

Generally, `std::unique_ptr` is used when you want your object to last only as long as a single owning reference to it does. Let's look at a practical example to demonstrate `std::unique_ptr` usage and some of its functions.

```
// big object declaration
class foo {
   public:
   void bar() { ... }
};

void processFoo(const foo& object) { ... }
```

First, we will create the smart unique pointer. Remember, do not use `new` operator on the unique pointer directly.

```
std::unique_ptr<foo> foo_ptr(new foo());
```

We can call the method on the object using the **->** operator:

```
foo_ptr->bar();
```

And pass the foo object reference to the function using the `*` operator. Note that the unique pointer cannot be copied or passed by value because it is a pointer.

```
processFoo(*foo_ptr);
```

It's possible to access the raw pointer using the `get()` method. It's especially helpful if you want to use the smart pointer to manage memory while still passing the raw pointer to code that doesn't support smart pointers.

```
foo_ptr.get();
```

We can also free memory before exiting the code block with a unique pointer declaration using the `reset()` method:

```
foo_ptr.reset();
```

std::make_unique

To make the creation of unique pointers easier and safer, the `std::make_unique` function constructs an object of a given type and wraps it in `std::unique_ptr`:

```
auto ptr = std::make_unique<int>(13);
```

This is also the preferred way to create unique pointers (instead of using the `new` operator). The only exception is when you need to customize a way to delete the object or are adopting a raw pointer from elsewhere —. In that case, do not use `std::make_unique`.

std::shared_ptr

Similar to `std::unique_ptr`, we will start with the C++ Reference definition of `std::shared_ptr`:

`std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object.

This means that `std::shared_ptr` is a smart pointer with **shared object ownership** semantics. It is worth mentioning that the shared pointer is destroyed when the remaining `std::shared_ptr` owning the object is destroyed.

Syntax

The shared pointer is declared as follows:

```
std::shared_ptr<int> ptr(new int); // allocation of new int on heap
```

The allocated int (or any other object within `std:shared_ptr`) is called a **managed object**. In contrast to the unique pointer, an object managed by a shared pointer can be shared with as many shared pointers as we like.

```
std::shared_ptr<int> ptr2 = ptr;
auto ptr3 = ptr;
```

Usage

Usually, you will use `std::shared_ptr` when you do want numerous references to your object and you don't want it to be deallocated until all of these references have been removed.

The methods shown for `std::unique_ptr` are the same for `std::shared_ptr`, like creation, calling object methods, dereferencing, accessing the raw pointer, and resetting it. In this part, we will focus only on those functionalities specific to `std::shared_ptr`.

Let's start with copy initialization and via assignment.

```
std::shared_ptr<int> ptr2(ptr);
std::shared_ptr<int> ptr3 = ptr;
```

It's also possible to check how many instances of `std::shared_ptr` manage the same object and if the current object is unique (i.e., other shared pointers don't manage this object):

```
ptr.use_count(); // returns number of shared pointers managing the same object as ptr
ptr.unique();    // returns true if ptr is the only shared_ptr managing object, false otherwise
```

Finally, the last functionality is the comparison operation. Two unrelated shared pointers will never be equal (even when they contain the same information), but related shared pointers are always equal.

```
std::shared_ptr<std::string> pt1(new std::string("str1"));
std::shared_ptr<std::string> pt2(new std::string("str1"));

std::cout << pt1 == pt2; // return false because pt1 and pt2 are not related

std::shared_ptr<std::string> pt3(pt1);

std::cout << pt1 == pt3; // returns true because pt1 and pt3 are related
```

std::make_shared

As in the case of `std::unique_ptr`, `std::shared_ptr` includes a dedicated (and preferred) method for creating pointers called `std::make_shared()`. It constructs an object of a given type and wraps it in `std::shared_ptr`:

```
auto ptr = std::make_shared<int>(13);
```

Be aware that there isn't a way to release the memory for the control block and the managed object separately when using `std::make_shared`. It creates a single heap allocation for both the control block and the managed object, so we have to wait until we can release both the managed object and the control block.

std::weak_ptr

As the C++ Reference defines:

`std::weak_ptr` is a smart pointer that holds a non-owning ("weak") reference to an object that is managed by `std::shared_ptr`. It must be converted to `std::shared_ptr` in order to access the referenced object.

Syntax

The weak pointer is declared as in the code below:

```
std::weak_ptr ptr;
```

And later it can be used to observe the object of a shared pointer:

```
auto sh_ptr = std::make_shared<int>(13)
ptr = sh_ptr; // watches the managed object of sh_ptr
```

Remember that a control block on a shared pointer object keeps track of the number of **shared and weak pointers**. The object is removed when the shared counter hits zero, but the control block remains active until the weak counter also reaches zero.

Usage

Why would we ever use a weak pointer? Generally, weak pointers are used when you do want to refer to your object from multiple places and do not want your object to be deallocated until all these references are themselves gone. Sometimes, you need to store the shared_ptr's underlying object without increasing the reference count. Often, this issue occurs when `shared_ptr` objects have cyclic references. Let's see an example:

```
struct A;

struct B {
   std::shared_ptr<A> A_ptr;
   ~B() { std::cout << "~B()"; }
};

struct A {
   std::shared_ptr<B> B_ptr;
   ~A() { std::cout << "~A()"; }
};

int main() {
   auto BB = std::make_shared<B>();
   auto AA = std::make_shared<A>();

   AA->B_ptr = BB;
   BB->A_ptr = AA;

   return 0;
}
```

The problem with the code above is that destructors will not be called so there is a memory leak. Keep in mind that the managed object of the shared pointer is deleted when the reference count reaches zero —. Let's analyze the situation. When `BB` goes out of scope, it will be not be deleted because it still manages the object to which `AA.B_ptr` points. A similar situation occurs with the `AA` —. If it goes out of scope, its managed object is not deleted either because `BB.A_ptr` points to it. This problem can be solved with a weak pointer.

```
struct A;

struct B {
   std::shared_ptr<A> A_ptr;
   ~B() { std::cout << "~B()"; }
};

struct A {
   std::weak_ptr<B> B_ptr; // using weak_ptr instead of shared_ptr
```

```
   ~A() { std::cout << "~A()"; }
};

int main() {
   auto BB = std::make_shared<B>();
   auto AA = std::make_shared<A>();

   AA->B_ptr = BB;
   BB->A_ptr = AA;

   return 0;
}
```

Now, both destructors are called when `BB` goes out of scope. It can be destroyed because a weak pointer pointed to it and later, `AA` can be destroyed because it is pointing to nothing. It doesn't matter whether `AA` or `BB` goes out of scope first. When `BB` goes out of scope, it calls for the destruction of all managed objects, like `A_ptr`. So, even if `AA` went out of scope first and was not destroyed, they will be destroyed together with `BB`.

Hash Tables

This chapter covers hash tables. You will learn the following:

1. What is a hash table and why should you use it?
2. What is hashing and a hash function?
3. What are the C++ standard library implementations for hash tables?

Introduction

A **hash table** is a data structure that stores elements in key-value pairs. **Key** is a unique value used to compute a table index and **value** is the data associated with the key. The benefit of using a hash table is its fast access time. Typically, the time complexity is a constant (O(1) access time).

Hashing Function

As stated previously, in a hash table, a new index is processed using the key and the corresponding value for this key is stored in the index. This process is called hashing and the function performing this operation is called a **hashing function**. For example, consider a hashing function as operation mod 10 and a set of key-value pairs to insert into hash table `{{15, 25}, {23, 63}, {12, 22}, {48, 78}, {30, 0}}`.

key	index based on hashing function	value
15	15 % 10 = 5	25
23	23 % 10 = 3	63
12	12 % 10 = 2	22
48	48 % 10 = 8	78
30	30 % 10 = 0	0

This means that the positions in the hash table will be the following:

position	0	1	2	3	4	5	6	7	8	9
value	0		22	63		25			78	

As you can see, there's a good chance that more than one key value will compute the same index. To solve this problem, in a standard library, values are stored in buckets — so multiple values can be under the same index.

C++ Standard Library Hash Tables

In the C++ standard library, four different hash tables are available. They are the equivalents of standard containers but with a hashing function:

- std::unordered_set (equivalent of std::set)
- std::unordered_multiset (equivalent of std::multiset)
- std::unordered_map (equivalent of std::map)
- std::unordered_multimap (equivalent of std::multimap)

Let's look at them one by one.

Set

Let's start with **std::unordered_set**. According to the C++ Reference:

`std::unordered_set` is an associative container holding a set of unique objects of type Key. Search, insertion, and removal have average constant time complexity.

The most important aspect of std::unordered_set is that it contains **unique objects**, which are of type **key**, and has average access time **constant**. An example of a `std::unordered_set` declaration is shown in the code below:

```
std::unordered_set<int> n;
```

To initialize `std::unordered_set`, we can simply assign values to it at declaration. Remember that the given values have to be unique.

```
std::unordered_set<int> n {0, 1, 2, 3, 4};
```

As in other standard containers, we can conduct operations like the following:

- size(),
- insert(...),
- erase(...),
- count(...),
- find(...),
- iteration over elements.

Similar to `std::unordered_set` is **std::unordered_multiset**. Let's start with a definition:

`std::unordered_multiset` is an associative container holding a set of possibly non-unique objects of type Key. Search, insertion, and removal have average constant time complexity.

This means that the only difference between `std::unordered_set` and `std::unordered_multiset` is that the latter allows multiple keys to be stored. An example of the `std::unordered_multiset` declaration is shown in code below:

```
std::unordered_multiset<int> n;
```

To initialize `std::unordered_multiset`, we can simply assign values to it at declaration. This time the values may be repeated.

```
std::unordered_multiset<int> n {0, 1, 2, 1, 2};
```

Map

Now, we will move to the map containers, starting with **std::unordered_map**. According to the C++ Reference:

`std:unordered_map` is an associative container holding key-value pairs with unique keys. Search, insertion, and removal of elements have average constant time complexity.

This means that the most important information about `std::unordered_map` is that it stores **key-value pairs**, where **key is unique** and the average access time is **constant**. The code below shows an example of a `std::unordered_map` declaration where key is of type `int` and value is of type `std::string`:

```
std::unordered_map<int, std::string> m;
```

To initialize `std::unordered_map`, we can simply assign values to it at declaration. Remember that the key values have to be unique.

```
std::unordered_map<int, std::string> m {{0, "zero"},
                                        {1, "one"},
                                        {2, "two"}};
```

Similarly, as with a set container, **std::unordered_multimap** and `std::unordered_map` have a lot in common. Let's look at the C++ Reference definition:

`std::unordered_multimap` is an unordered associative container that supports equivalent keys (an unordered_multimap may contain multiple copies of each key value) and that associates values of another type with the keys. (...) Search, insertion, and removal have average constant time complexity.

The only difference is that `std::unordered_multimap` allows for keys to be repeated. The code below shows an example of a `std::unordered_multimap` declaration where key is of type `int` and value is of type `std::string`:

```
std::unordered_multimap<int, std::string> m;
```

To initialize `std::unordered_multimap`, as before, we can assign values to it at declaration. This time the keys don't need to be unique.

```
std::unordered_multimap<int, std::string> m {{0, "zero"},
                                             {1, "one"},
                                             {2, "two"},
                                             {0, "zero"}};
```

And of course, it supports several operations like other standard library containers.

Summary

To summarize this module, we would like to compare all of the standard library associative containers.

Container	Sorted	Value	Identical keys possible	Average access time
std::set	yes	no	no	logarithmic
std::unordered_set	no	no	no	constant
std::map	yes	yes	no	logarithmic
std::unordered_map	no	yes	no	constant
std::multiset	yes	no	yes	logarithmic
std::unordered_multiset	no	no	yes	constant

Container	Sorted	Value	Identical keys possible	Average access time
std::multimap	yes	yes	yes	logarithmic
std::unordered_multimap	no	yes	yes	constant

std::array Container

This chapter covers the `std::array` container. You will learn the following:

1. What is std::array and why was it created?
2. What is the syntax of `std::array`?

Introduction

Let's start with the definition.

std::array is a container that is built on top of fixed-size arrays. It combines the performance and accessibility of a C-style array with other benefits of a standard container such as:

- knowing its own size
- supporting assignment
- random access iterators

It also supports standard container operations like `std::sort`, `std::find`, etc.

`std::array` is similar to a C-style array but not identical. When we pass a C-style array to a function, we have to pass the address and the size of the array. The reason is that the information about the array size gets lost when passing the array address to the function (i.e., in form of pointer). This problem can be solved using std::array. To use std::array, remember to include `<array>` STL.

This datatype is especially important when using SYCL as it doesn't allow dynamic memory allocation in a kernel.

Syntax of std::array

An example of a `std::array` declaration is shown in the code below:

```
std::array<int, 5> n;
```

We need to specify the datatype that will be stored in the array (in the example, it is `int`) and its size (`5` in the example code).

To initialize an `std::array`, we can simply assign values to it at declaration.

```
std::array<int, 5> n = {0, 1, 2, 3, 4};
std::array<int, 5> m {0, 1, 2, 3, 4};
```

As the length of the array needs to be known at compile time, we can declare the array and later initialize it with values.

```
std::array n;
n = {0, 1, 2, 3, 4};
```

Usage

We can use `std::array` in every situation when we are using a C-style array. Just make sure you know its size in advance.

To access the element at a specific position, we can use C-style brackets `[ ]` (because the std::array elements are placed side by side in memory) or the `at()` method. The only difference between them is that `at()` checks bound, while using the C-style brackets doesn't check bound.

```
std::array n {1, 2, 3, 4, 5};
n[3];    // returns element of the array at position 3, doesn't check bound
n.at(3); // returns element of the array at position 3, checks bound
```

We can also access the first and last element of the array using `front` and `back`:

```
n.front(); // returns first element of the array
n.back();  // returns last element of the array
```

To get the length of the `std::array`, use the `size()` method:

```
n.size();
```

Like other standard containers, the `std::array` provides iterator functions that allow it to iterate over the container in a standard or reversed way.

Multidimensional Array

Like in C-style arrays, it's possible to create a multidimensional `std::array`. Let's look at the example with a 5x3 `std::array`:

```
std::array<std::array<int, 5>, 3> n {
  {0, 1, 2, 3, 4},
  {0, 1, 2, 3, 4},
  {0, 1, 2, 3, 4}
};
```

This means that as a datatype for the outer array, we are using another array.

When using SYCL, consider not using an array of arrays. Instead, use a dedicated structure that contains arrays. It will improve memory optimization when storing and accessing the elements. What is more, it will be more readable for other developers.

Move Semantics

This chapter covers move semantics. You will learn the following:

1. What are move semantics?
2. What are the different value categories and when should they be used?
3. What is universal reference T&&?
4. How and why should std::move be used?
5. How is a move constructor created? (Rule of Five)

Introduction

Move semantics allows an object under certain conditions to take ownership of some other object's external resources.

Value Categories (glvalue and rvalue)

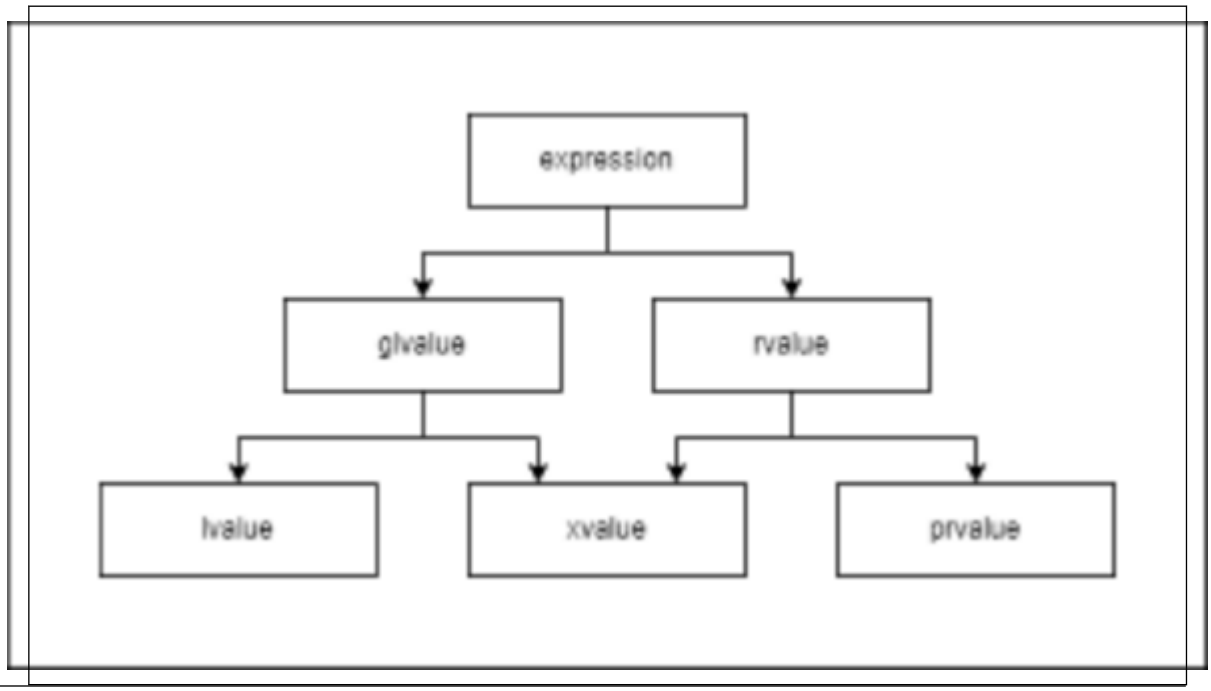
In C++, every expression has a type and belongs to a specific **value category**. These are the basic rules for a compiler to follow when creating, copying, and moving objects during expression evaluation.

Here are some C++ expression value categories:

- **glvalue** — Expression that has an identity; it's possible to determine if two expressions refer to the same entity.

- **rvalue** — Expression that can be moved from.
- **lvalue** — Has an identity and can't be moved from.
- **xvalue** — Has an identity and can be moved from.
- **prvalue** — Doesn't have an identity and can be moved from.

The diagram below shows different expression types and the dependencies between them:



Let's look at the lvalue and rvalue in the following example:

```
size_t x = 0;
x = 1; // x expression is lvalue

size_t foo() { /* ... */ }

foo(); // Result of foo call expr is an rvalue
```

Any name of variable, function, template parameter object, or data member is an lvalue. It's important to note that it doesn't matter how complex the expression is. As long as it maintains an identity, the expression is an lvalue.

The integer constants are *prvalues*, like in the code above — the result of a function call.

Functions Returning lvalue and prvalue

Let's imagine a function returning a `int` value:

```
int returnValue() {
   return 3;
}
```

In this case, `returnValue()` returns the temporary number `3`, which is a prvalue. Now, we will try to assign the value to it:

```
returnValue() = 17; // error
```

We will receive an error: `lvalue required as left operand of assignment`. That's because we are trying to use the left operand of the assignment on prvalue. But when we change the `returnValue()` function to return a reference to an already existing memory location, everything will work fine:

```
int globalValue = 43;

int& returnValue() {
   return globalValue;
}

// ...

returnValue() = 17; // works fine
```

Even though the ability to return an lvalue may not seem intuitive, it can be useful when implementing more advanced functions like overloaded operators.

lvalue-to-prvalue Conversion

An lvalue may be converted to a prvalue. This is totally legal and occurs frequently. Let's look at `+operator` as an example. According to the C++ standard, it takes two prvalues as arguments and returns a prvalue.

```
int x = 10, y=20;
int z = x + y;
```

`x` and `y` are lvalues, but the additional operator wants prvalues. *How is it possible?* Because of an implicit lvalue-to-prvalue conversion. There are many more operators performing similar conversions. But what about the opposite — converting prvalue to lvalue? It is not possible due to the C++ design.

Universal References (&&)

One of the main features related to the rvalues introduced in C++11 was rvalue reference. Usually, the `&&` notation is known as a syntax for rvalue reference. But it is not always true. `T&&` can hold both lvalue and rvalue references, which is called a **universal reference**. But remember that `&&` only means a universal reference when type deduction is involved. In other cases, we can assume that it means only an rvalue reference. Let's see it in code. We will start with a universal reference, as the `T` is deduced.

```
template<typename T>
void foo(T&& param);
```

Now, let's move on to an rvalue reference, as there is no type deduction.

```
void foo(std::string&& param);
```

Finally, the last thing is to show the concept of prefect forwarding, which is when a universal reference can be propagated, preserving the l-r 'valueness'.

```
template<typename T>
void foo(T&& param) { /* ... */ }

template<typename T>
void bar(T&& param) {
   foo(std::forward<T>(param)); // l or r value depending on the param passed to `bar`
}
```

In this case, because both functions `foo` and `bar` are using a universal reference, `foo` will receive an l or r value, depending on the param passed to `bar`.

std::move

Let's start by answering the question: What is `std::move`?

According to the C++ Reference:

`std::move` is used to indicate that an object t may be "moved from"
(i.e., allowing the efficient transfer of resources from t to another object).
In other words, it is a way to efficiently transfer contents of an object to another,
leaving the source in a valid but undefined state. When you move a value from a register
or memory location to another place, the value on the source register or memory location
is still there. And more formally, `std::move` is a C++ Standard Library function
that's defined in the `<utility>` header. It is used to cast an l-value reference
to an r-value reference, which enables move semantics.

Let's see an example. We will start with a declaration of the function consuming the element.

```
void consume_element(std::unique_ptr<int> element);
```

Then, let's declare it and consume using a prepared function and `std:move`.

```
std::unique_ptr<int> element = std::make_unique<int>(30);
consume_element(std::move(el));
```

After those operations, the declared element `element` is nullptr, as it was moved.

```
assert(element == nullptr);
```

Move Constructor and Rule of Five

`std::move` is actually just a request to move. If the type of the object does not have a move constructor/assign operator defined, the move operation will fall back to a copy. In that case, we will not experience any benefits of using the move operation.

That is why it is important to know how to create a move constructor. At the same time, in C++ we have something called **Rule of Five**, which is as follows:

1. If a class requires a user-defined destructor, a user-defined copy constructor, or a user-defined copy assignment operator, it almost certainly requires all three.
2. Any class for which move semantics are desirable needs to declare the move constructor and the move assignment operator.

Those result in creating five elements:

1. user-defined destructor
2. user-defined copy constructor
3. user-defined copy assignment operator
4. user-defined move constructor
5. user-defined move assignment operator

Let's show this with an example. Imagine a class called `MoveClass` with a private member called `str_ptr` being `char*`. To show the Rule of Five, we need to declare the following:

- custom destructor
- custom copy constructor
- custom move constructor
- custom copy assignment operator
- custom move assignment operator

```
class MoveClass {
   char* str_ptr;

public:
   explicit MoveClass(const char* s = "") : str_ptr(nullptr) {
      if (s) {
         std::size_t size = std::strlen(s) + 1;
```

```
         str_ptr = new char[size];      // allocate
         std::memcpy(str_ptr, s, size); // populate
      }
   }

   // Destructor - we need to deallocate str_ptr
   ~MoveClass() {
      delete[] str_ptr;
   }

   // Copy constructor - uses explicit constructor, parameter passed is const&
   MoveClass(const MoveClass& other)
      : MoveClass(other.str_ptr) {}

   // Move constructor - uses std::exchange function, parameter passed is &&
   MoveClass(MoveClass&& other) noexcept
      : std_ptr(std::exchange(other.str_ptr, nullptr)) {}

   // Copy assignment operator - uses copy constructor,
   // passed parameter similarly to copy constructor is const&
   MoveClass& operator=(const MoveClass& other) {
      return *this = MoveClass(other);
   }

   // Move assignment operator - uses std::swap function,
   // passed parameter similarly to copy constructor is &&
   MoveClass& operator=(MoveClass&& other) noexcept {
      std::swap(str_ptr, other.str_ptr);
      return *this;
   }
};
```

Lambdas

This chapter covers lambdas. You will learn the following:

1. What is a C++ lambda expression?
2. What are the parts of a lambda?
3. How can a lambda expression be used in C++ code?

Introduction

C++ lambda expression — often called a lambda — allows us to define anonymous function objects (functors) which can be used inline or passed as an argument. This was introduced in C++11 as a more convenient and concise way for creating anonymous functors. Let's take a look at a few simple lambda examples:

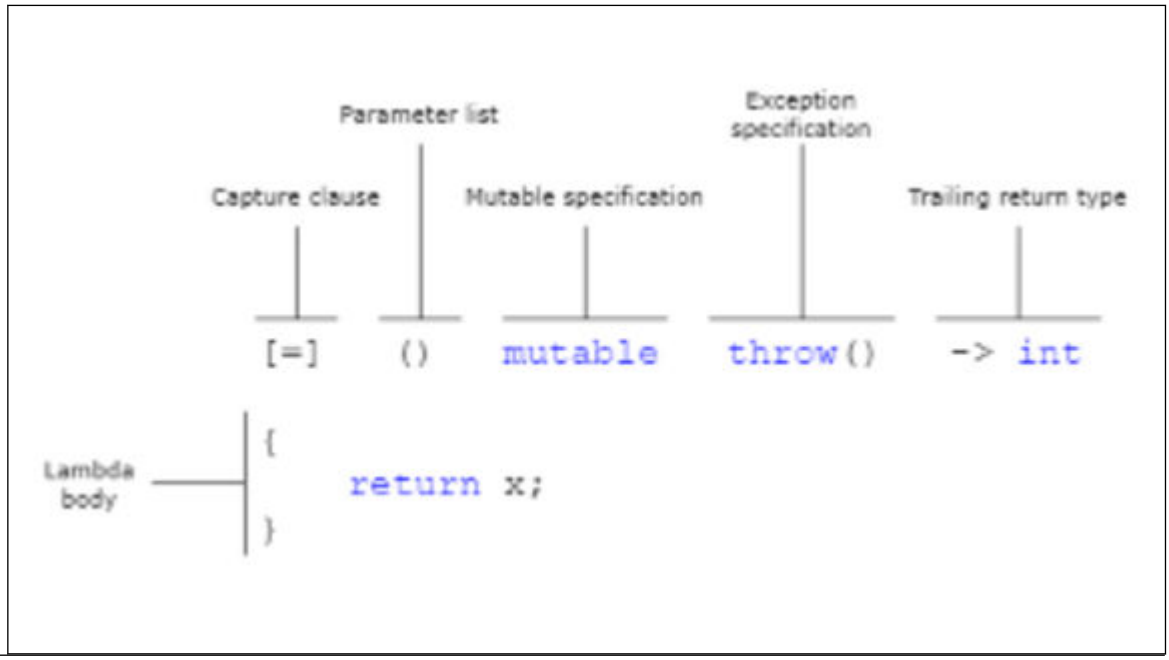
```
// Simple lambda printing string
[] { std::cout << "Simple lambda\n"; };

// Simple lambda returning sum of two elements
auto lambda_sum = [](int x, int y){ return x + y; };

// Simple lambda for sorting elements of the vector using std::sort
std::vector<int> vec{23, 1, 2, 44, 56, 37, 4, 9, 0};
std::sort(vec.begin(), vec.end(), [](int a, int b) { return a > b; });
```

Parts of the Lambda Expression

There are six basic parts of a lambda expression:

1. capture clause
2. parameter list — *optional*
3. mutable specification — *optional*
4. exception specification — *optional*
5. trailing-return-type — *optional*
6. lambda body

The image below shows them in a practical example.



Capture Clause

Capture clause is also called **capture list** or **lambda-introducer**. It is the beginning of the lambda expression. It specifies which variables are captured and whether the capture is by value or by reference. Examples of the capture clause are as follows:

- `[]` — capture nothing
- `[=]` — capture local objects like variables and parameters by value
- `[&]` — capture local objects like variables and parameters by reference
- `[this]` — capture `this` by reference
- `[a, &b]` — capture objects `a` by value and `b` by reference

Let's see them in an example.

```
int number = 7;

// Lambda returning vale of number by using [=]
auto getNumber =  [=] { return number; };
getNumber(); // returns 7

// Lambda returning sum of number and passed argument as a value by using [=]
auto addNumbers = [=](int y) { return number + y; };
addNumbers(6); // returns 13 (7+6=13)
```

```
// Lambda returning reference to the number by using [&]
auto getReference = [&]() -> int& { return number; };
getReference(); // returns int& to number
```

There is also a feature called **generalized capture**. We can use this feature to introduce and initialize new variables in the capture clause, without the need to have those variables exist in the lambda function's enclosing scope. The type of the variable is deduced from the type produced by the expression. This feature is useful when we want to use it in lambda move-only variables from the surrounding scope:

```
auto ptr = std::make_unique<int>(8);

// Lambda using generalized capture
auto lambda = [capturedValue = std::move(ptr)] { /* use ptr*/ };
```

Parameter List

Lambdas can capture variables and accept *input parameters*. A **parameter list** is optional and, in most aspects, resembles the parameter list for a function. Let's see the same simple code in a function form and as a lambda expression.

```
int add(int x, int y) {
   return x + y;
}

auto lambdaAdd = [](int x, int y) { return x + y; };
```

In lambdas, it's possible to use the `auto` keyword as the type specifier in a parameter list if the type is generic. It can also take another lambda expression as an argument.

```
auto lambdaAdd = [](auto x, auto y) { return x + y; };
```

Mutable Specification

By default, value captures cannot be modified inside the lambda because the compiler-generated method is marked as `const`, but using the `mutable` keyword cancels this out. This means that the **mutable specification** enables the body of a lambda expression to modify variables that are captured by value.

```
int number = 7;

// number is reference, so the lambda modifies original
auto lambdaReference = [&number] { number = 2; };

// Error - lambda can perform const-only operations on number
auto lambdaValue = [number] { number = 2; };

// Due to usage of mutable lambda can modify number
auto lambdaMutable = [number] () mutable { number = 2; };
```

Exception Specification

You can specify that the lambda will not throw any exception using the `noexcept` keyword. You can see what will happen if you compile and run the following code:

```
[]() noexcept { throw 13; } ;
```

Most C++ compilers should show the warning during compilation, but other than that, the code will not throw the exception.

Return Type

In general, the returned type of the lambda expression is automatically deduced and there is no need to use the `auto` keyword for that, as shown below:

```
[]() { std::cout << "Sample output.\n"; }; // deduced type of the lambda is void
```

You can specify **trailing-return-type**, which resembles the return-type part of the standard function. But please remember that it must follow the parameter list (even if it is empty) and you must use the `->` keyword before the return-type.

```
// lambda returning int as trailing-return-type specifies
[]() -> int { return 13; };
```

You can omit the return-type part of a lambda expression if the lambda body contains just one return statement or if the expression doesn't return a value.

```
// lambda returning int as deduced type from the single return statement
[](int x) { return x; }(7);
```

Lambda Body

Because the lambda expression is the same as an ordinary function, its body can contain anything that's allowed in a function body. This means that a lambda body, similar to a function body, can access the following:

- Captured variables from the enclosing scope
- Parameter
- Locally declared variables
- Class data members (when lambda is declared inside a class and `this` is captured)
- Variables with static storage duration (like global variables)

Let's look at a code example. We would like to print the elements of the declared vector together with information that includes whether the number is even or odd. The vector declaration is as follows:

```
std::vector<int> v {1, 2, 3, 4};
```

Now, we can prepare the function printing number and the phrase "is even" or "is odd."

```
// lambda returning int as deduced type from the single return statement
void isEvenOrOdd(int n){
   std::cout << n;

   if (n % 2 == 0) {
      std::cout << " is even\n";
   } else {
      std::cout << " is odd\n";
   }
}
```

To apply this function to all vector elements, we are using the `for_each` function from the algorithms library.

```
for_each(v.begin(), v.end(), isEvenOrOdd);
```

The same result can be achieved by using a lambda expression instead of the `isEvenOrOdd` function:

```
for_each(v.begin(), v.end(), [](int n) {
   std::cout << n;

   if (n % 2 == 0) {
      std::cout << " is even\n";
   } else {
      std::cout << " is odd\n";
   }
});
```

As you can see, there are no limitations related to the size of the lambda. The only limit may be the readability of the code, as lambdas are typically used as small helper functions.

Type Inference in C++ (auto and decltype)

This chapter covers type inference. You will learn:

1. What is type reference in C++?
2. How should the keywords `auto` and `decltype` be used?
3. What are the differences between `auto` and `decltype`?

Introduction

Type inference refers to the automatic deduction of the data type of an expression in a programming language. Since C++11, many keywords were included that allow type inference to be left to the compiler. Before C++11, each data type needed to be *explicitly declared* at compile time. Now, using type inference, we can avoid wasting time writing out things that the compiler already knows or can deduce. As all the types are deduced in the compiler phase only, the compilation time increases slightly; at the same time, it doesn't affect the runtime of the created program. In C++, two keywords are introduced for type inference:

- `auto` keyword
- `decltype` type specifier

`auto` Keyword

The `auto` keyword indicates the type of declared variable to be deduced automatically from its initializer. If the return type of a function is `auto`, it will be evaluated at runtime by the returned type expression. It's important to note that the variable declared with the `auto` keyword needs to be initialized at declaration time. Otherwise, a compilation error occurs. Let's see an example of `auto` usage:

> **NOTE** In these examples, we are using `typeid` to get the type of the variables.

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

int main(){
   auto x = 1;
   auto y = 3.7;

   // checking type of declared variables
   std::cout << typeid(x).name() << '\n';
   std::cout << typeid(y).name() << '\n';

   return 0;
}
```

The output of the code above is:

```
i
d
```

This means that variable `x` is type of `int` when variable `y` is type of `double`. Using `auto` helps avoid long initializations (e.g., when creating iterators for containers).

`decltype` Type Specifier

The `decltype` specifier inspects the declared type of an entity or the type of an expression. We can say that `decltype` is more like an operator that evaluates the type of passed expression.

When you use `auto`, you declare a variable with a particular type, but when using `decltype` you extract the type from the variable. Let's see an example of `decltype` usage:

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

int foo() { return 13; }

int main(){
   // x has the same type as a type returned by foo function
   decltype(foo()) x;

   // checking type of x
   std::cout << typeid(x).name();

   return 0;
}
```

The output of this code is:

```
i
```

This means that variable `x` is type `int`.

Now, let's see an example using both `auto` and `decltype`:

```
#include <iostream>
#include <vector>

int main(){

   std::vector<int> vec(10);

   // using auto for type deduction
   for(auto i = vec.begin(); i < vec.end(); i++){
      std::cin >> *i;
   }

   // using decltype for type deduction
   for(decltype(vec.begin()) i = vec.begin(); i < vec.end(); i++){
      std::cin >> *i;
   }

   return 0;
}
```

In this example, we are using `auto` and `decltype` for the same purpose — deduction of the iterator type.

> **NOTE** The type denoted by `decltype` can be different from the type deduced by `auto`.

Summary

It's important to understand that `auto` and `decltype` serve different purposes, so they are not exactly the same. `auto` is used for automatic type deduction whereas `decltype` yields the type of a specified expression. Unlike `auto`, which deduces types based on values being assigned to the variable, `decltype` deduces the type from an expression passed to it.

Initializers in if and switch Statements

This chapter covers initializer statements in if and switch blocks. You will learn the following:

1. What is an initializer statement?
2. How can you benefit from using it inside the if-else and switch blocks?
3. What is the syntax of initializer statements in if and switch blocks?

Introduction

There are many situations when we need to check the value returned by a function or perform conditional operations based on the value of the function. Let's look at the example pseudocode below:

```
// some function declaration
int foo(/*...*/)

// function call and return value storing in variable
int var = foo(/*...*/);

if (var == /*some value*/){
   // do something
} else {
   // do something else
}
```

There is nothing wrong with this code, but it can be improved in certain situations. As shown in the example, the variable `var` is used only for this if-else statement. At the same time, it leaks into the surrounding scope. In that case, we need to make sure to not mix it up with other variables. Also, if the compiler can explicitly know that this variable will only be used there, it may potentially produce more optimized code.

Syntax

Let's look at the syntax of the if-else and switch blocks.

Initializer Statement Outside the Conditional Blocks

You may notice that if-else conditional blocks and switch blocks have a generic format. First, we have an initialization of a variable that will be checked, and then there is a conditional block using this initialized variable. For an if-else block, it usually looks like this:

```
// initializer-statement

if (condition){
   // do something
} else {
   // do something else
}
```

And for switch block, similarly, like this:

```
// initializer-statement

switch(variable){
   // cases
}
```

Initializer Statements in if and switch Blocks

Starting with C++17, it is possible to initialize a variable inside if and switch statements:

```
if (initializer-statement; condition){
   // do something
} else {
   // do something else
}
```

```
switch(initializer-statement; variable){
   // cases
}
```

Summary

Initializers in if and switch statements allow the variable to be assigned to the scope of this statement. Note that using this language feature can result in more complex code when initialization and comparison are separate concerns.

Standard Template Library (STL) on Concurrent and Parallel Algorithms

This chapter covers concurrent and parallel STL algorithms. You will learn the following:

1. What are STL algorithms?
2. What execution policies are available as part of the STL?
3. How are different execution policies used?
4. What are the benefits and drawbacks of using specific execution policies?
5. What are the best practices when using concurrent and parallel STL algorithms?

Introduction

The STL offers over 150 algorithms including those that run sequentially. To take advantage of parallel hardware, programmers must parallelize these libraries using low-level threading APIs, which is not easy. Fortunately, since C++17, most of the STL algorithms are parallelized and support for vectorization has been added.

What are STL Algorithms?

STL algorithms are a set of powerful functions that enable the execution of various operations on data containers such as vectors, lists, and arrays. These algorithms are designed to be generic, making them ideal for a wide range of applications. The STL helps avoid repeatedly implementing simple functions, and instead focus on addressing higher-level problems. Examples of algorithms available within the STL include `std::find`, `std::sort`, and `std::replace`. These algorithms can drastically improve the efficiency and readability of the code.

Execution Policies

To use the STL parallel algorithms, you need to include the `<execution>` header. The next step is to invoke the chosen algorithm with an execution policy, which allows you to specify how the algorithm should be executed. In C++, the following execution policies are available:

- sequenced_policy (and the corresponding global object `std::execution::seq`) — sequential execution of the algorithm.
- parallel_policy (and the corresponding global object `std::execution::par`) — parallel execution of the algorithm.
- parallel_unsequenced_policy (and the corresponding global object `std::execution::par_unseq`) — parallel execution of the algorithm with the ability to use vector instructions.
- unsequenced_policy (and the corresponding global object `std::execution::unseq`) — execution of the algorithm with the ability to use vector instructions.

Let's see examples of the use of these policies.

> **NOTE** We will be using small data sets just to show how the policies work, but remember that the benefits of using policies usually comes from dealing with large amounts of data.

sequenced_policy

We will use the `std::sort` algorithm to demonstrate the sequential execution policy:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <execution>

int main(){
   // container creation with unsorted elements
   std::vector<int> vec {7, 13, 4, 2, 56, 24, 1};

   // using std::sort on the prepared container vec with sequential execution policy
   std::sort(std::execution::seq, vec.begin(), vec.end());

   for(int i : vec) {
      std::cout << i << ' ';
   }
   return 0;
}
```

Using sequenced_policy has several advantages:

- It is ideal for small tasks that don't warrant the parallel overhead.
- Data races can be avoided.
- It is simple and predictable.

At the same time, sequenced_policy is not efficient for large tasks with a lot of data.

parallel_policy

We will use the `std::find` algorithm to demonstrated the parallel execution policy:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <execution>

int main(){
   // container creation
   std::vector<int> vec {7, 13, 4, 2, 56, 24, 1};

   // using std::find on the prepared container vec with parallel execution policy
```

```
   auto it = std::find(std::execution::par, vec.begin(), vec.end(), 4);

   if(it != vec.end()){
       std::cout << *it;
   }
   return 0;
}
```

When using parallel_policy, we can benefit from the following:

- Faster execution for larger tasks and on larger datasets
- Optimal usage of multicore systems

At the same time, it is important to remember that:

- It may introduce overhead. If that's the case, it is not always faster than sequential execution.
- It is the programmer's responsibility to avoid data races and deadlocks.

parallel_unsequenced_policy

For parallel_unsequenced_policy, we used the `std::transform` algorithm with a prepared lambda function that returns a number squared. It's important to understand that the result can be every permutation of {1, 4, 9, 16, 25} as the operations are performed nonsequentially.

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <execution>

int main(){
   // container creation
   std::vector<int> vec {1, 2, 3, 4, 5};

   // using std::transform on the prepared container vec
   // with parallel unsequential execution policy
   std::transform(std::execution::par_unseq,
     vec.begin(),
     vec.end(),
     vec.begin(),
     [](int x){ return x*x; });

   for(int i : vec){
       std::cout << i << ' ';
   }
   return 0;
}
```

Here, similar to the parallel_policy, we can:

- Realize faster execution for repetitive operations
- Benefit from using hardware with vector instructions

On the other hand, we need to remember that it is not suitable for all tasks (e.g., tasks where the order of operations is important).

unsequenced_policy

For the last policy — unsequenced_policy — we are using the `std::for_each` algorithm:

```
#include <algorithm>
#include <iostream>
#include <vector>
```

```
#include <execution>

int main(){
   // container creation
   std::vector<int> vec {1, 2, 3, 4, 5};

   // using std::for_each on the prepared container vec
   // with unsequential execution policy
   std::for_each(std::execution::unseq,
     vec.begin(),
     vec.end(),
     [](int x){ std::cout << x << ' ';});

   return 0;
}
```

Using unsequenced_policy, we can:

- Experience fast execution on a single thread
- Avoid race conditions

At the same time, the execution of the sequence is not deterministic, so we need to make sure that the order is not important for the task we are doing.

Best Practices

To make the most of STL on concurrent and parallel algorithms, it's worth considering the following best practices:

1. **Choose the right algorithm** — The STL provides many different algorithms, so it is important to choose the right one and select the appropriate execution policy. Ensure that you fully understand the problem you want to solve.
2. **Profile your code** — It can be useful to profile your code to identify the performance bottlenecks. Remember that you can achieve the best results by optimizing the critical sections of the code.
3. **Minimize shared data** — Minimizing the amount of shared data allows you to reduce the chances of race conditions and synchronization overhead.

Summary

STL algorithms and concurrent programming approaches are useful additions to data scientists and software engineers' toolkits. Try it yourself and create the code that compares the performance of different execution policies on the same task. Remember to work on a large set of data.

Nested Namespaces

This chapter covers C++ namespaces. You will learn the following:

1. What is a namespace?
2. How can you declare your own namespace?
3. How do nested namespaces work in C++?

Introduction

The **namespace** is a declarative region that provides scope to the identifiers like names of types, functions, and variables. Namespaces are usually used to organize the code into logical groups and to avoid name collisions. It can be especially important when you're using different libraries. One of the examples of the namespace scope is the C++ Standard Template Library (STL), where all the classes, methods, and

templates are declared. You can find STL namespace in the code `std::` before multiple class or function declarations from STL in C++. Sometimes you can also find the directive :code:`using namespace std;.

Defining a Namespace

To define a namespace, you have to start with the `namespace` keyword followed by the name:

```
namespace your_namespace_name {
   // ...
   // declarations
   // ...
}
```

Note that there is no semicolon `;` after the closing bracket. To call the function or variable declared within the namespace, prepend the namespace name as shown below:

```
your_namespace_name::function_name(/*...*/);
your_namespace_name::variable_name;
```

Nested Namespaces

In C++, namespaces can be nested, which makes it possible to define one namespace inside another. The resolution of namespace variables is hierarchical. The syntax is shown below:

```
namespace A {
   // ...
   // declarations
   namespace B {
   // ...
   // declarations
      namespace C {
      // ...
      // declarations
      }
   }
}
```

As you can see, those declarations take a lot of space and sometimes it's difficult to track all the levels of the declared namespaces. But in modern C++, writing nested namespaces can be simplified:

```
namespace A::B::C {
      // ...
      // declarations
}
```

It is useful when you want to declare functions and variables in the deepest namespace. It also allows you to easily track the namespace in which you're writing your declarations. Similarly, as with a single namespace, to call the function or variable declared within the nested namespace, you need to write all levels of the namespaces:

```
A::B::C::function_name(/*..*/);
A::B::C::variable_name;
```

Now, let's see it in the real example:

```
#include <iostream>

// outer namespace declaration
namespace outer {
   void foo() {
      std::cout << "Outer foo() function call. \n";
```

```
   }

   // inner namespace declaration
   namespace inner {
      void foo() {
         std::cout << "Inner foo() function call. \n";
      }
   }
}


int main() {
   outer::inner::foo();
   outer::foo();

   return 0;
}
```

The output of the code will look like this:

```
Inner foo() function call.
Outer foo() function call.
```

This means that the inner foo function was called first, followed by the outer foo function.

4) SYCL Implementations of Modern C++

Introduction

Vector addition is a fundamental operation in parallel computing, often serving as a basic building block for more complex algorithms. It involves adding corresponding elements of two vectors to produce a resulting vector. **When implemented efficiently, vector addition can significantly enhance the performance** of various applications, ranging from scientific simulations to real-time graphics rendering. As technology advances, heterogeneous computing using platforms such as CPUs and GPUs has become increasingly prevalent, prompting the need for efficient and portable programming models.

Traditionally, **the conventional approach of heterogeneous systems programming required developers to manage memory explicitly, leading to intricate and error-prone code.** However, with the advent of Unified Shared Memory (USM) in SYCL, the process has become streamlined. USM enables the creation of memory regions that are accessible across different devices. Therefore, **USM eliminates the need for explicit data transfers between host and device memories.** This simplifies memory management and accelerates development that can drastically enhance both code readability, portability, and performance. Programmers can use USM to focus more on algorithmic design and less on memory-related intricacies.

In this exploration, we delve into two approaches for performing vector addition: the conventional method with explicit host-device data transfer and the modern USM-enabled approach using SYCL. We'll examine the steps involved in both techniques, highlighting their individual strengths and limitations.

Conventional vector_add with SYCL

In the following code, we dissect the conventional vector addition technique that uses `buffer`, `handler` and `accessor`. This approach typically requires explicit memory management, including allocating memory on the host and devices, copying data between them, and synchronizing their operations:

```
//==============================================================
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================

#include <CL/sycl.hpp>

using namespace sycl;

int main() {
   const int N = 256;

   // STEP 1 : Initialize vectors
   std::vector<int> vector1(N, 10);
   std::vector<int> vector2(N, 20);
   std::vector<int> vectorR(N,  0);

   // STEP 2 : Create buffer for the vectors
   buffer vector1_buffer(vector1);
   buffer vector2_buffer(vector2);
   buffer vectorR_buffer(vectorR);

   // STEP 3 : Submit task to add vector
   queue q;
   q.submit([&](handler &h) {

      //# STEP 4 : Create vectors accessors
      accessor vector1_accessor (vector1_buffer, h, read_only);
      accessor vector2_accessor (vector2_buffer, h, read_only);
      accessor vectorR_accessor (vectorR_buffer, h);

      //# STEP 5 : Proceed with the calculation
      h.parallel_for(range<1>(N), [=](id<1> index) {
         vectorR_accessor[index] = vector1_accessor[index] + vector2_accessor[index];
      });

   });

   // STEP 6 : Create a host accessor to copy data from device to host
   host_accessor h_a(vectorR_buffer,read_only);

   // STEP 7 : Print output values
   std::cout<<"\nOutput Values: ";
   for (int i = 0; i < N; i++) std::cout<< h_a[i] << " ";
   std::cout<<"\n";
```

```
   return 0;
}
```

These are the main SYCL steps to read from the code:

- **Vector Initialization:** Three vectors `vector1`, `vector2`, and `vectorR` of size N are created on the host and initialized with specific values. They represent the input vectors and the result vector of a vector addition operation.
- **Create Buffers:** Three SYCL buffers, `vector1_buffer`, `vector2_buffer`, and `vectorR_buffer`, are created to hold the data of the vectors. These buffers are used for data transfer between the host and the SYCL device (e.g., GPU).
- **Submit SYCL Task:** A SYCL queue q is created, and a task is submitted to the queue using the `q.submit()` method. This task will execute on a SYCL device, such as a GPU.
- **Create Accessors:** Accessors are created for the input and output buffers. `vector1_accessor` and `vector2_accessor` are read-only accessors, while `vectorR_accessor` is a regular accessor.
- **Parallel For:** A parallel loop is launched using `h.parallel_for()`, which will execute the vector addition operation in parallel across multiple work items. Each work item is responsible for adding one element of the vectors. **Note that this function uses the accessors, not the buffers or the initial vectors.**
- **Host Accessor:** After the SYCL task is completed, a host accessor `h_a` is created for the `vectorR_buffer` to copy the data from the device to the host.

While effective, this method can lead to intricate code and hinder productivity due to its manual memory management requirements. In the next section, we'll implement vector_add using USM.

USM vector_add with SYCL

In this discussion, we implement vector addition using Unified Shared Memory (USM), highlighting how the SYCL programming model simplifies the utilization of heterogeneous computing resources. The main takeaway is that with USM there is no requirement to define `buffers`, `handlers`, and `accessors`. In this case, the USM `malloc_shared` function is sufficient to allocate and manage memory access:

```
//==============================================================
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================

#include <CL/sycl.hpp>

using namespace sycl;

static const int N = 256;

int main() {
   queue q;

   // STEP 1 : Get device information
   std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

   // STEP 2 : Initialize vectors in shared memory
   int *data1 = malloc_shared<int>(N, q);
   int *data2 = malloc_shared<int>(N, q);
   int *dataR = malloc_shared<int>(N, q);

   // STEP 3 : Asign values to vectors
   for (int i = 0; i < N; i++) {
      data1[i] = 10;
      data2[i] = 20;
```

```
      dataR[i] =  0;
   }

   // STEP 4 : Proceed with the calculation
   q.parallel_for(range<1>(N), [=](id<1> i) {
      dataR[i] = data1[i] + data2[i];
   }).wait();

   // STEP 5 : Print output values
   for (int i = 0; i < N; i++) std::cout << dataR[i] << " ";
   std::cout << "\n";

   // STEP 6 : Release memory
   free(data1, q);
   free(data2, q);
   free(dataR, q);

   return 0;
}
```

This part of the code:

```
std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";
```

gets device information from the `q`, and prints information about the selected device.

USM gets established here:

```
int *data1 = malloc_shared<int>(N, q);
int *data2 = malloc_shared<int>(N, q);
int *dataR = malloc_shared<int>(N, q);
```

where three shared memory arrays `data1`, `data2`, and `dataR` are allocated using `malloc_shared<int>(N, q)`. They are accessible during computation by both the host and device. USM also offers `malloc_device` and `malloc_host` to explicitly allocate memory on the host or device, but they will not be covered in this course.

So, when should you use USM versus the conventional SYCL? In summary, using USM is a good approach to consider when:

- The code has many pointers that are susceptible to future changes.
- When it is expected that the application will gain complexity over time.

SYCL matrix_multiplication Using USM

Matrix multiplication is a fundamental operation in linear algebra. It plays a crucial role in various scientific and engineering applications, such as image processing, simulation, and machine learning. The operation involves taking the dot product of rows from one matrix with the columns of another matrix to produce a resulting matrix. While conceptually simple, efficient computation of matrix multiplication becomes a challenge when dealing with large matrices due to the inherent parallelism that can be leveraged for optimization. This is where SYCL comes into play. SYCL is a programming model and API that enables developers to write code that can be executed on a variety of heterogeneous platforms, including CPUs, GPUs, FPGAs, and other accelerators, using a single-source approach.

In this context, leveraging SYCL for matrix multiplication offers the potential to harness the power of various computing devices while maintaining a unified codebase. **By expressing parallelism with SYCL, developers can achieve optimized matrix multiplication routines that efficiently distribute computation across different devices.** In the code below we present the matrix multiplication code in SYCL format using USM:

```
//==============================================================
// Copyright © Intel Corporation
//
// SPDX-License-Identifier: MIT
// =============================================================

#include <CL/sycl.hpp>

using namespace sycl;

static const int N = 3;

int main() {
    queue q;

    // STEP 1 : Get device information
    std::cout << "Device : " << q.get_device().get_info<info::device::name>() << "\n";

    // STEP 2 : Initialize vectors in shared memory
    int *data1 = malloc_shared<int>(N * N, q);
    int *data2 = malloc_shared<int>(N * N, q);
    int *dataR = malloc_shared<int>(N * N, q);

    // STEP 3 : Asign values to vectors
    for (int i = 0; i < N * N; i++) {
        data1[i] = 10;
        data2[i] = 20;
        dataR[i] =  0;
    }

    // STEP 4 : Proceed with the calculation
    q.parallel_for(range<2>{N,N}, [=](item<2> item){
        const int j = item.get_id(0);
        const int i = item.get_id(1);

        for (int k = 0; k < N; ++k)
            dataR[i*N+j] += data1[i*N+k] * data2[k*N+j];
    }).wait();

    // STEP 5 : Print output values
    for (int i = 0; i < N * N; i++) std::cout << dataR[i] << " ";
    std::cout << "\n";

    //# STEP 6 : Release memory
    free(data1, q);
    free(data2, q);
    free(dataR, q);

    return 0;
}
```

This program produces the following output:

```
Device : Intel(R) UHD Graphics [0x9a60]
600 600 600 600 600 600 600 600 600
```

Summary

Using buffers, handlers, and accessors versus using USM (Unified Shared Memory) in SYCL each has its own set of advantages and disadvantages. Here's a comparison of the two approaches:

Advantages of Using Buffers, Handlers, and Accessors:

- **Fine-Grained Control:** Buffers, handlers, and accessors provide fine-grained control over memory management and data transfer. This can be advantageous when you need precise control over memory allocation, data movement, and synchronization. It can also be more flexible when multiple devices need to access the same data.
- **Explicit Data Movement:** Buffers and accessors allow you to explicitly control when data is transferred between the host and device. This can be useful when optimizing data movement for specific application requirements.
- **Data Safety:** Accessors help enforce data safety by ensuring that data is accessed and modified within the appropriate scope and access mode, reducing the risk of data races and synchronization issues.
- **Parallelism:** Handlers and accessors facilitate the creation of parallel tasks and dependencies, enabling you to express complex parallelism patterns in your code.
- **Device Selection:** Buffers and accessors can be used to specify and control the choice of the device for computation, allowing you to target specific hardware.

Disadvantages of Using Buffers, Handlers, and Accessors:

- **Complexity:** Managing buffers, handlers, and accessors can add complexity to your code, especially for applications that require extensive data movement and synchronization.
- **Verbose Code:** SYCL code using buffers, handlers, and accessors can be verbose, requiring more lines of code compared to USM for simple operations.
- **Learning Curve:** Beginners may find it challenging to grasp the concepts of buffers, handlers, and accessors, particularly if they are new to parallel programming and SYCL.

Advantages of Using USM:

- **Simplicity:** USM simplifies memory management by providing a unified memory space accessible by both the host and device. It reduces the need for explicit memory allocation and synchronization, resulting in simpler and more concise code.
- **Ease of Use:** USM is easier to learn and use, making it an attractive option for developers new to SYCL or parallel programming in general.
- **Implicit Data Movement:** With USM, data movement between the host and device is managed implicitly, reducing the need for manual data transfers and synchronization points.
- **Reduced Boilerplate:** USM reduces the amount of SYCL-specific code needed to express parallelism and manage memory, leading to less boilerplate in your application.

Disadvantages of Using USM:

- **Limited Control:** While USM simplifies memory management, it also limits your control over low-level memory operations. This may be a disadvantage in scenarios where fine-grained control is essential for performance optimization.
- **Performance Trade-offs:** Depending on the hardware and use case, USM may introduce performance tradeoffs compared to using buffers and explicit data movement control.
- **Compatibility:** Not all SYCL-compatible devices and platforms support USM, so there may be limitations on where you can deploy USM-based code.

In summary, the choice between using buffers, handlers, and accessors versus USM in SYCL depends on your specific application requirements and your familiarity with SYCL programming. Buffers, handlers, and accessors provide more control but can be more complex, while USM simplifies memory management but may not offer the same level of control in all situations. Consider the tradeoffs and choose the approach that best suits your application's needs and your development expertise.

5) Next step: SYCL Essentials

Introduction to SYCL Essentials

Congratulations on embarking on your first SYCL journey with "SYCL 101"!!, your gateway to the world of SYCL programming. As you delve into the foundational aspects of SYCL, you're building a strong base that will empower you to explore the possibilities of parallel programming.

Now that you have completed the SYCL 101 course, **why stop here?** Would you like to elevate your SYCL skills? Join **SYCL Essentials!**

SYCL Essentials is the next FREE step in your SYCL education. As you're starting to grasp the basics, you'll soon realize the incredible potential SYCL holds for optimizing code performance, harnessing parallelism, and unleashing the full power of modern hardware. You will have access to the most advanced Intel® CPUs and GPUs, and you will learn how to target them with your code! SYCL Essentials is a more advanced course that has been meticulously crafted to take your skills to the next level and open doors to a deeper understanding of SYCL.

What Awaits You in SYCL Essentials?

Module 1: oneAPI Intro: Dive into the comprehensive oneAPI ecosystem and its significance in creating efficient, high-performance code across diverse hardware architectures.

Module 2: DPCPP Program Structure: Uncover advanced program structuring techniques that facilitate code organization and maintainability, ensuring your SYCL programs are not only powerful but also elegant.

Module 3: DPCPP Unified Shared Memory: Learn how to seamlessly share data between devices to improve data access efficiency and enhance performance across your applications.

Module 4: DPCPP Sub Groups: Explore the intricacies of sub-groups, a vital concept in SYCL that enables more fine-grained control over work-item execution, leading to optimized parallel processing.

Module 5: Intel® Advisor: Master the use of Intel Advisor to analyze and fine-tune your SYCL code, ensuring you're making the most of your hardware's capabilities...and that's just the beginning. From efficient memory management to advanced debugging tools, from enhanced data manipulation to leveraging sub-group capabilities, each module in "SYCL Essentials" is designed to equip you with the skills needed to tackle complex programming challenges head-on.

Why Should You Take the Leap?

Unlock Unprecedented Performance: SYCL Essentials will empower you to create code that not only runs but thrives on modern hardware. Unleash the full potential of parallelism and witness your applications running faster and smoother than ever before.

Stay Ahead of the Curve: The tech landscape is evolving rapidly, and mastering advanced SYCL concepts will position you as a trailblazer in the field of parallel programming. Be at the forefront of innovation.

Dive Deeper into Optimization: SYCL Essentials arms you with the tools to fine-tune your code, optimize memory access patterns, and squeeze every ounce of performance from your applications.

Are You Ready to Take the Next Step?

If "SYCL 101" has sparked your curiosity and ignited your passion for SYCL programming, "SYCL Essentials" is the natural progression that will catapult your skills to new heights.

In the next sections, we present descriptions for the modules along with their learning objectives more in detail. Take a look at them and get ready to tackle complex challenges, write code that's not just functional but exceptional, and become a true SYCL expert!

Enroll in "SYCL Essentials" today and embark on a journey that promises to transform you into a proficient, confident, and innovative SYCL programmer. Your future in parallel programming awaits, seize the opportunity now. It is totally FREE!!

Start SYCL Essentialshere.

Module 1: oneAPI Intro

This module introduces you to oneAPI and oneAPI's implementation of SYCL (Data Parallel C++/DPC++). You will learn about the basic SYCL hello world program and how to compile SYCL code for Intel hardware using Data Parallel C++ Compiler.

Learning Objectives

- Explain how the oneAPI programming model can solve the challenges of programming in a heterogeneous world.
- Use oneAPI projects to enable your workflows.
- Understand the SYCL language and programming model.
- Familiarize yourself with the use Jupyter Notebooks for training throughout the course.

Module 2: DPCPP Program Structure

This module demonstrates basic **SYCL code anatomy** and program structure. You will learn about important **SYCL classes** and different types of kernels. You will also learn about **buffer accessor memory model** and synchronization when using buffers.

Learning Objectives

- Explain the SYCL fundamental classes.
- Use device selection to offload kernel workloads.
- Decide when to use basic parallel kernels and ND Range kernels.
- Create a host accessor.
- Build a sample SYCL application through hands-on lab exercises.

Module 3: DPCPP Unified Shared Memory

This module demonstrates how to implement different modes of **Unified Shared Memory (USM)** in SYCL code, as well as how to solve for data dependencies for in-order and out-of-order tasks.

Learning Objectives

- Use new SYCL2020 features such as USM to simplify programming.
- Understand implicit and explicit ways of moving memory using USM.
- Solve data dependency between kernel tasks optimally.

Module 4: DPCPP Sub-Groups

This module examines SYCL **sub-groups** and why they are important. The code samples demonstrate how to implement a query for sub-group info, sub-group shuffles, and sub-group algorithms.

Learning Objectives

- Understand the advantages of using sub-groups in SYCL.
- Take advantage of sub-group algorithms for performance and productivity.
- Use sub-group shuffle operations to avoid explicit memory operations.

Module 5: Intel® Advisor

This module demonstrates various aspects of **Intel® Advisor**. The first uses Intel Advisor to show performance offload opportunities found in a sample application, followed by additional command-line options for getting offload advisor results. The second gives an example of **roofline analysis** and command-line options for getting Advisor results.

Learning Objectives

- Show how Intel Advisor can help you decide which part of the code should or should not be offloaded to the accelerator.
- Run Offload Advisor and generate an HTML report.
- Read and understand the metrics in the report.
- Get a performance estimation of your application on the target hardware.
- Decide which loops are good candidates for accelerator offload.

Module 6: VTune™ Profiler

This module demonstrates using **VTune™ Profiler on the command line** to collect and analyze gpu_hotspots. You will learn how to collect performance metrics and explore the results with the HTML output rendered inside of the notebook. This module is meant for exploration and familiarization. It does not require any code modification.

Learning Objectives

- Profile a SYCL application using the VTune profiling tool on Intel® DevCloud
- Understand the basics of VTune command-line options for collecting data and generating reports.

Module 7: DPCPP Library

This module demonstrates using **Intel® oneAPI DPC++ Library (oneDPL)** for heterogeneous computing. You will learn how to use various Parallel STL algorithms for heterogeneous computing and also look at a gamma-correction sample code that uses oneDPL.

Learning Objectives

- Explain oneDPL Parallel API Algorithms with examples.
- Explain oneDPL Parallel API Iterators with examples.
- Explain oneDPL Extension API Utility classes with examples.

Module 8: DPCPP Reduction

This module demonstrates how reductions can be parallelized. You will learn about the **reduction group algorithm** and **kernel reductions in SYCL**.

Learning Objectives

- Understand how reductions can be performed with parallel kernels.
- Take advantage of the reduce group function to make reductions at the sub_group and work_group level.
- Use the reduction object to simplify reduction with parallel kernels.
- Use multiple reductions in a single kernel.

Module 9: DPCPP Buffers And Accessors In Depth

This module demonstrates various ways to create **buffers in SYCL**. You will learn how to use **sub-buffers**, buffer properties, and when to use_host_ptr, set_final_data, and set_write_data. You will also learn about accessors, host_accessors, and their use cases.

Learning Objectives

- Explain buffers and accessors in depth.
- Understand how to create and use sub-buffers.
- Explain buffer properties and when to use_host_ptr, set_final_data, and set_write_data.
- Explain accessors and the modes of accessor creation.
- Explain host accessors and their different use cases.

Module 10: DPCPP Graphs Scheduling Data Management

This module demonstrates how to utilize USM, buffers, and accessors to apply memory management and take control over **data movement implicitly and explicitly** to help utilize different types of data dependencies that are important for ensuring the execution of graph scheduling. You will also learn to select the correct modes of **dependencies in graphs scheduling**.

Learning Objectives

- Utilize USM, buffers, and accessors to apply memory management and take control over data movement implicitly and explicitly.
- Utilize different types of data dependencies that are important for ensuring the execution of graph scheduling.
- Select the correct modes of dependencies in graphs scheduling.

Module 11: Intel® Distribution for GDB

This module demonstrates how to use Intel® Distribution for GDB to debug kernels running on GPUs.

Learning Objectives

- Show how Intel Distribution for GDB can help you debug GPU kernels.
- Run Intel Distribution for GDB.
- Understand inferiors, threads, and SIMD lanes as shown in GDB.
- Use different methods to examine local variables for different threads and lanes.

Module 12: DPCPP Local Memory And Atomics

This module demonstrates how to utilize a device's **shared local memory** to reduce latency in accessing data for kernel computation. The module also introduces **atomic operations**, which help avoid data race conditions when multiple work items are updating the same memory object.

Learning Objectives

- Use local memory to avoid repeated global memory access.
- Understand the usage of group barriers to synchronize all work items.
- Use atomic operations to perform reduction.