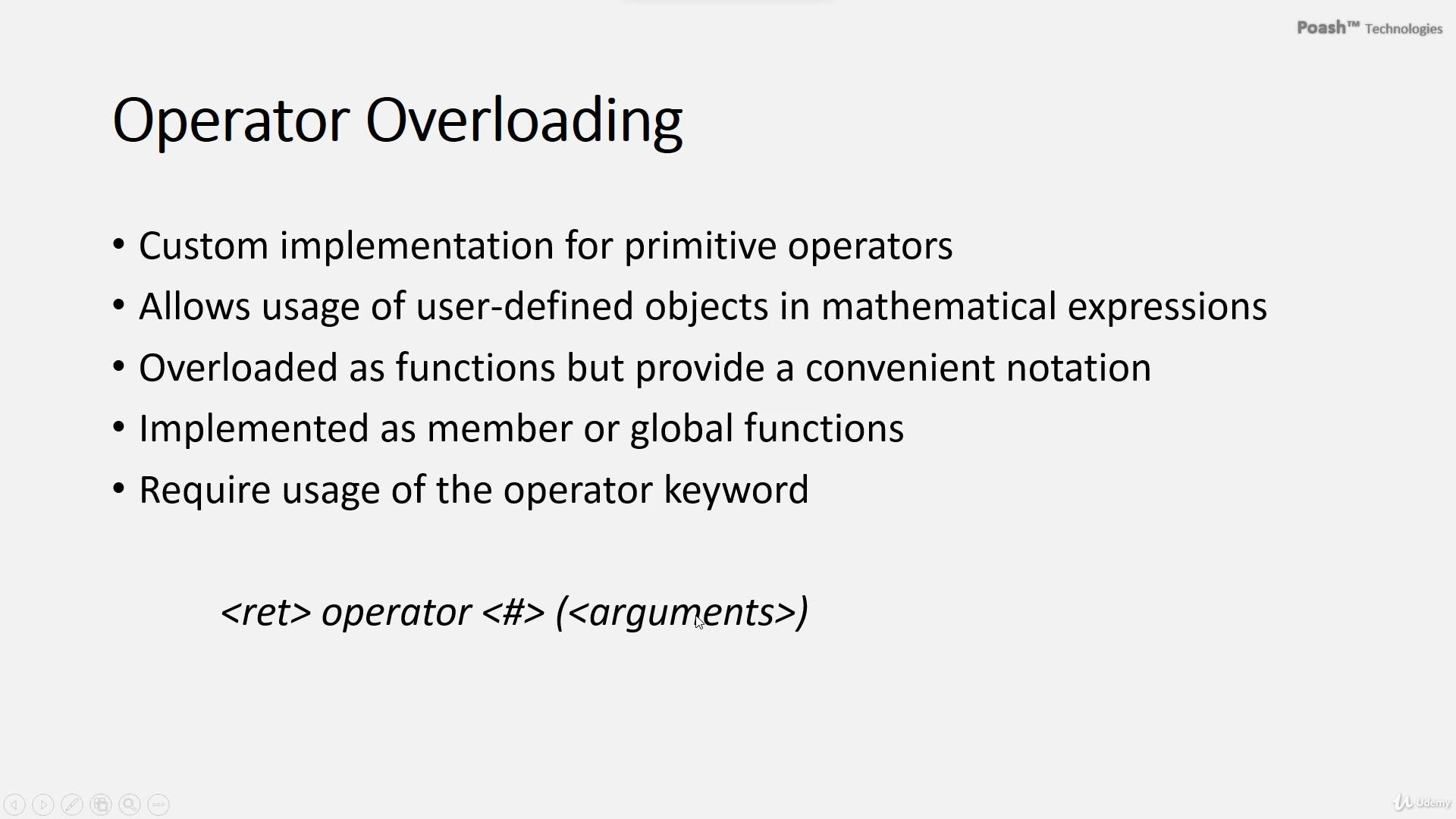
## 6. Operator Overloading





# Class “+” overloading

A screen shot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

* This plus operator will invoke a function call
* If any function returns \*this, that means it is an opportunity to return by reference and no temporary will be created.
* That is why the usage of the pre-increment operator is more efficient than the post-increment because the post-increment operator requires the creation of a temporary object.

A screenshot of a computer code

Description automatically generated

# The assignment operator.

* If we do not overload the assignment operator. A compiler will provide a default implementation and that would perform a shallow copy.

A screenshot of a computer program

Description automatically generated

# Global overloading

A computer screen shot of text

Description automatically generated

* To not use .getvalue Function on sum, overloading operator <<, in iostream, and. Setvalue function

A screenshot of a computer program

Description automatically generated

<< sum1 intead of << sum1.getvalue

1. **Definition**: Operator overloading allows custom implementations of primitive operators for user-defined objects, enabling their use in complex mathematical expressions.
2. **Implementation**: Overloaded operators can be implemented as either global functions or member functions, with the **operator** keyword indicating that they are overloaded operators.
3. **Syntax**: The syntax for an overloaded operator includes the return type, the operator being overloaded, and the operands as arguments.
4. **Member Function Overloading**: When overloading as a member function, one operand is passed implicitly through the **this** pointer.
5. **Return Type**: Overloaded operators typically return a value of the same type as the operands.
6. **Pre-Increment and Post-Increment Operators**: These operators are overloaded differently to distinguish between them. Pre-increment returns the incremented value, while post-increment returns the original value before incrementing.
7. **Efficiency**: Pre-increment is more efficient than post-increment because it avoids the creation of a temporary object.
8. **Comparison Operator Overloading**: The overloading the comparison operator to check for equality between objects.

The importance of overloading the assignment operator (**operator=**) in a custom class, particularly when dealing with dynamic memory allocation.

1. **Assignment Operator Importance**: When objects of a class are assigned to each other, the assignment operator is invoked. If this operator is not explicitly overloaded, the compiler generates a default implementation, which performs a shallow copy of member variables.
2. **Shallow Copy Issue**: Shallow copying can lead to problems, especially when dynamic memory allocation is involved. Without a custom assignment operator, the default behavior may cause memory leaks or undefined behavior.
3. **Custom Implementation**: To prevent shallow copying and manage resources properly, a custom assignment operator should be provided. This operator typically deallocates memory for the current object and then allocates new memory to store the values from the assigned object.
4. **Self-Assignment Check**: A crucial aspect of the custom assignment operator is to check for self-assignment (assigning an object to itself). Without this check, deleting the current object's memory before assigning values can lead to undefined behavior.
5. **Rule of 5**: To adhere to best practices in resource management, the class should also implement the move constructor and move assignment operator, in addition to the copy constructor, destructor, and assignment operator. This ensures proper handling of resource ownership and efficiency when objects are moved.

A computer code with text

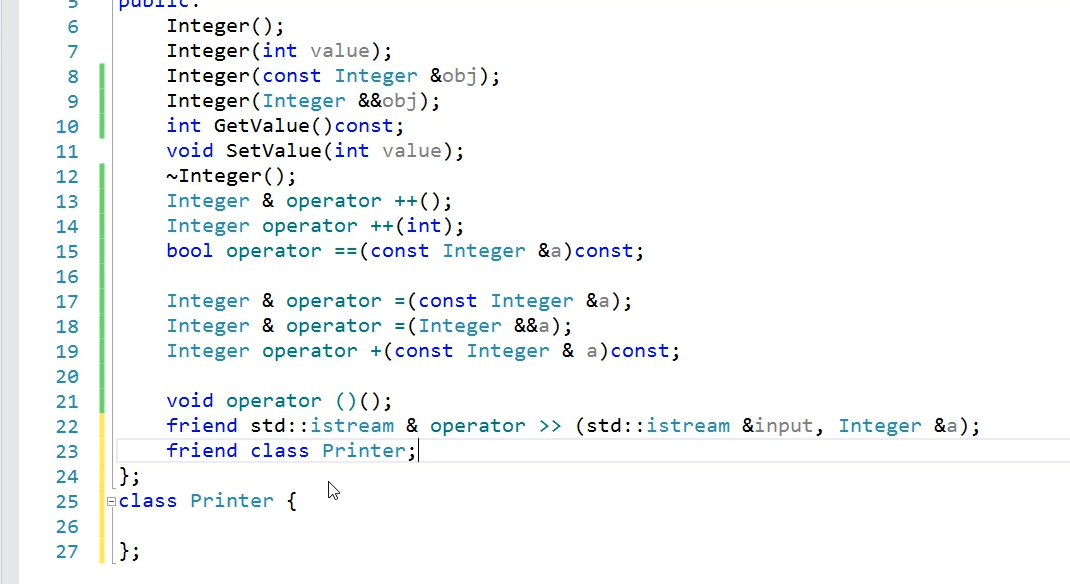
Description automatically generated

A computer screen shot of a code

Description automatically generated

# Keyword friend

* If a function wants to access private members in class
* That is why C++ provides the keyword friend. Using the keyword friend, We can make a function a friend of a class, that function will then have access to all the members of the class whether they are private, protected, or public.
* Can also make a class as a friend of another class.

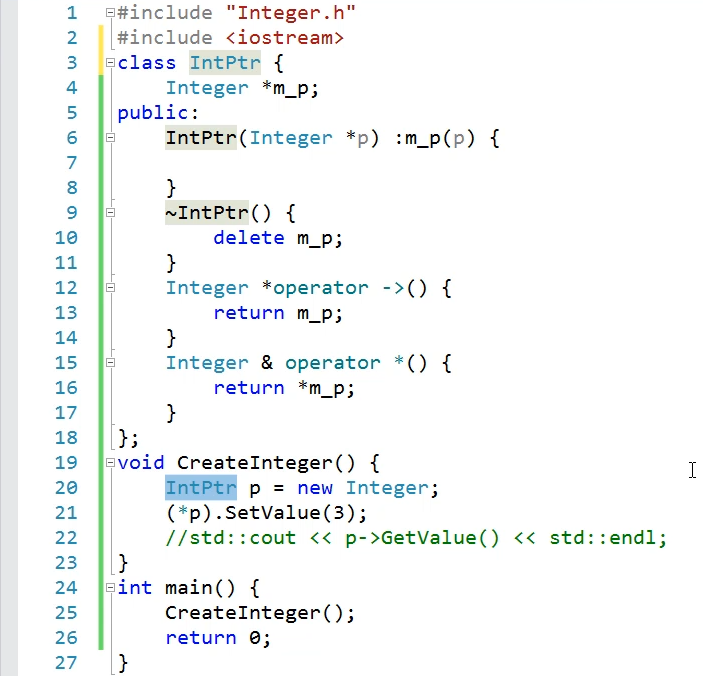


the use of the **friend** keyword in C++ to allow functions or classes access to private members of a class:

1. **Need for Access to Private Members**: Sometimes, when overloading operators or performing certain operations, it's necessary to access private members of a class directly.
2. **Using friend Keyword**: C++ provides the **friend** keyword to grant access to private members. By declaring a function or class as a friend inside another class, it gains access to all members of that class, regardless of their access level (private, protected, or public).
3. **Declaration Syntax**: To declare a function or class as a friend, it's preceded by the **friend** keyword in the class declaration.
4. **Friend Functions**: A function declared as a friend can access private members of the class it's a friend of. Omitting the **friend** keyword would make the function a member of the class, losing the intended access.
5. **Friend Classes**: Similarly, a class can be declared as a friend of another class, granting it access to the private members of that class.
6. **Caution in Usage**: While **friend** provides flexibility, it's discouraged to use it extensively. Direct access to private members can lead to bugs and make the code less maintainable. It's recommended to use **friend** only when no other viable solution exists.

In summary, the **friend** keyword in C++ allows controlled access to private members of a class, but its usage should be limited to situations where no better alternative is available to solve a particular problem.

Top of Form



Now with the asterisk operator, I can even dereference this object like a pointer.

# Smart pointers

* So, all the places where we use a pointer can be replaced with this object that uses the RAII idiom.
* This object p is a local object but it behaves like a pointer. And it also automatically deletes the underlying memory, so we can say this object is a smart pointer.
* A smart pointer behaves like a pointer but it automatically frees the memory. Unfortunately, this smart pointer can be used only with Integer objects but C++ standard library provides us with smart pointers that can be used to manage any kind of pointer.
* C++ provides different kinds of smart pointers and they are defined in the header file memory.
* So smart pointer will handle the deletion of the memory allocated and does not need to call delete

the concept of smart pointers in C++, illustrating it through code examples. Here's a breakdown:

1. **Manual Memory Management**: Initially, the code demonstrates manual memory management using raw pointers. It creates an **Integer** object on the heap, sets its value, and displays it. However, manual deletion of the pointer (**delete**) is prone to errors and memory leaks.
2. **Resource Acquisition is Initialization (RAII)**: The code introduces RAII, an idiom where the lifetime of a resource is bound to the lifetime of a local object. This ensures automatic resource cleanup when the local object goes out of scope, mitigating the need for explicit memory management.
3. **Implementation of Smart Pointer**: To implement RAII for dynamic memory allocation, the code defines a class **IntPtr** that encapsulates a raw pointer to an **Integer** object. The **IntPtr** class automatically deallocates the memory in its destructor.
4. **Overloading Operators**: The code overloads the arrow (**->**) and dereference (**\***) operators for the **IntPtr** class to mimic pointer behavior. This allows accessing the member functions and dereferencing the underlying pointer seamlessly.
5. **RAII in Action**: By using the **IntPtr** object, memory allocation is automatically managed, and the **Integer** object's destructor is called when the **IntPtr** object goes out of scope, ensuring proper cleanup.
6. **Smart Pointer Definition**: Finally, the code defines the **IntPtr** object as a smart pointer because it exhibits pointer behavior while automatically managing memory. It contrasts this custom implementation with smart pointers provided by the C++ standard library, which offer broader functionality and can manage any pointer type.

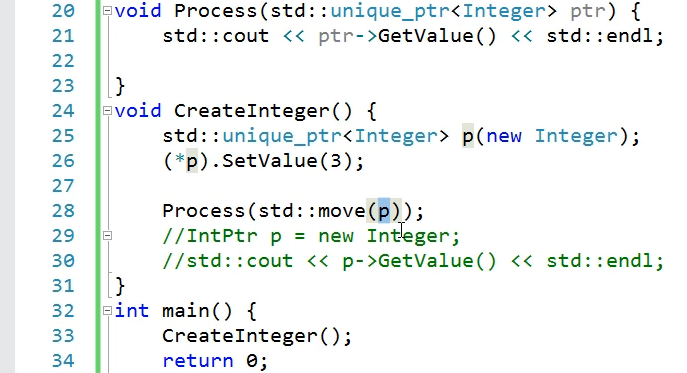
In summary, smart pointers in C++ automate memory management by associating resource cleanup with object lifetimes, thus preventing memory leaks and simplifying memory management tasks.

# The unique smart pointer

* The first smart pointer is the unique\_ptr. This is a class template and it requires the type of pointer that you want to manage.



* A unique pointer(unique\_ptr) is used when you do not want to share the underlying resource.
* That means we can not create a copy of the unique pointer. If you try to create a copy of the unique pointer, its copy constructor is a deleted function.
* Even though you cannot create a copy of the unique pointer. You can move it, it has move semantics only and does not support copy semantics.



**In this Example:**

* If we try to invoke the Process(p); function like this, this will not compile because you're trying to pass the unique pointer(unique\_ptr) by value and that would create a copy.
* It's important to create a unique pointer in the createinteger() function because you are going to initialize it and then you want to pass it to some other function like Process().
* This is not going to work but you can move the ownership of this resource from this unique pointer to the one in this function and that can be accomplished by calling the library function, move().

# The shared smart pointer

* A shared pointer is used when you want to share the underlying resource with other parts of the code.
* It internally implements some kind of reference counting and each time a copy of a shared pointer is created, The reference count is incremented by one.
* When a shared pointer is destroyed, the reference count is decremented and if the reference count becomes zero then it releases the underlying resource, so we can still use the shared pointer p after the Process() function.

A screen shot of a computer code

Description automatically generated

**Note:**

Modern C++ emphasizes the use of smart pointers rather than raw pointers.

the usage and characteristics of different types of smart pointers in C++, namely **unique\_ptr** and **shared\_ptr**. Let's break down the key points:

1. **unique\_ptr**:
   * Defined in the **<memory>** header, **unique\_ptr** is used to manage pointers to dynamically allocated objects.
   * It signifies exclusive ownership of the managed object, meaning it cannot be shared or copied. Attempting to copy a **unique\_ptr** will result in a compilation error.
   * It supports move semantics, allowing ownership transfer to another **unique\_ptr**.
   * The excerpt demonstrates the usage of **unique\_ptr** to manage memory allocation and automatically deallocate it when the **unique\_ptr** goes out of scope.
2. **shared\_ptr**:
   * Similar to **unique\_ptr**, **shared\_ptr** is used for managing dynamic memory but with shared ownership semantics.
   * Multiple **shared\_ptr** instances can share ownership of the same dynamically allocated object.
   * Internally, **shared\_ptr** maintains a reference count, incrementing it when a copy is made and decrementing it when a copy is destroyed.
   * The memory associated with a **shared\_ptr** is deallocated only when the reference count drops to zero, indicating no more shared ownership.
   * This type of smart pointer is suitable for scenarios where multiple parts of the code need access to the same dynamically allocated object.
3. **Comparison**:
   * **unique\_ptr** is suitable when exclusive ownership is required, and ownership transfer might occur.
   * **shared\_ptr** is used when shared ownership is needed, allowing multiple parts of the code to access and manage the same resource.
   * The excerpt provides examples of using both **unique\_ptr** and **shared\_ptr** and demonstrates their respective behaviors and advantages.
4. **Best Practices**:
   * Modern C++ encourages the use of smart pointers over raw pointers to prevent memory leaks and improve memory management.
   * Smart pointers automate memory management tasks, reducing the likelihood of errors associated with manual memory allocation and deallocation.
   * By adopting smart pointers, developers can write safer and more robust code, minimizing the risk of memory leaks and resource mismanagement.

In summary, the excerpt highlights the importance of using smart pointers in modern C++ programming to ensure efficient and reliable memory management, ultimately leading to more stable and maintainable codebases.

# Operator overloading Rules

A white background with black text

Description automatically generated

* Examine some rules associated with operator overloading. When you overload any operator, The compiler ensures that its basic behavior does not change.
* That means the order in which it evaluates the arguments, the precedence of the operators, and the number of operands that an operator accepts.
* All these do not change. Overloaded operator functions should be non-static except for new and delete. New and delete are special operators and are overloaded only in special circumstances especially when you want to control how memory is allocated and deallocated.
* An operator that accepts more than one argument should always have one of the operands as a user-defined type.
* Otherwise, the compiler will not allow you to overload it. If a binary operator accepts the first argument as a primitive type then it has to be overloaded as a global function.
* C++ does not allow us to overload all the operators. The following operators and some other operators are not allowed to be overloaded such as member access. (dot) operator, ternary operator(?:), a pointer to a member operator(.\*), sizeof, hash(#), and some casting operators. These are not allowed to be overloaded by the language.
* The reason that there is no overloading is because it will confuse the language or may cause certain basic features to stop functioning. C++ allows us to overload only predefined operators.
* You cannot define any new operators or overload them. When you overload any operator, you should overload it to mimic the conventional behavior only. Otherwise, that would confuse the users.

Here are some rules associated with operator overloading based on the provided text:

1. **Basic Behavior Preservation**:
   * When overloading any operator, the compiler ensures that its basic behavior does not change. This includes the order of evaluation of arguments, operator precedence, and the number of operands the operator accepts.
2. **Non-Static Functions**:
   * Overloaded operator functions should generally be non-static, except for **new** and **delete**, which are special operators and are overloaded only in specific circumstances, such as controlling memory allocation and deallocation.
3. **At Least One User-Defined Type**:
   * If an operator accepts more than one argument, at least one of the operands should be a user-defined type. This requirement allows the compiler to distinguish between overloaded operators and built-in operators.
4. **Global Function Overloading**:
   * If a binary operator accepts the first argument as a primitive type, it must be overloaded as a global function rather than a member function.
5. **Restricted Operators**:
   * Certain operators cannot be overloaded, including the member access (dot) operator **.**, ternary operator **?:**, pointer to member operator **.\***, **sizeof**, **#** (hash), and some casting operators. These restrictions exist to prevent confusion and ensure the proper functioning of basic language features.
6. **Predefined Operators Only**:
   * C++ only allows overloading of predefined operators. It does not permit the definition or overloading of new operators.
7. **Conventional Behavior**:
   * Overloaded operators should mimic the conventional behavior expected by users. Deviating from this behavior can lead to confusion.

Following these rules helps ensure that operator overloading in C++ is used appropriately and does not introduce unexpected behavior or confusion for users.

# Type Conversions

A white background with black text

Description automatically generated

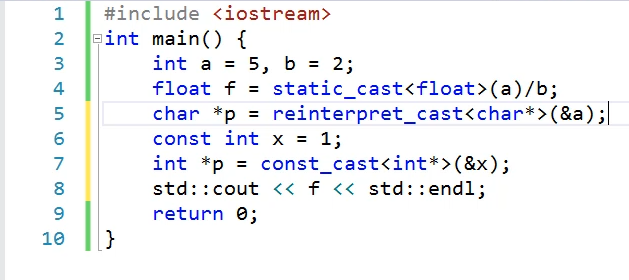
* C-style casts are discouraged in C++ because they do not check for the validity of the cast.

# Static cast.

* Static cast if you try to build with casts between different types., there's an error, cannot convert from int\* to char\*.

# Reinterpret cast

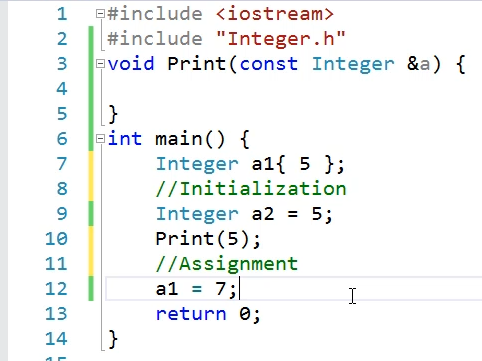
* A reinterpret\_cast allows casting between different types. But the advantage of reinterpret cast is that if there are any qualifiers on the source then they are not discarded.
*  C-style casts will discard the qualifiers.
* This will not give an error in the C-cast



Type conversion, also known as casting, is the process of converting one data type into another. This can be done implicitly by the compiler or explicitly by the programmer using casting operators:

1. **Implicit Type Conversion**:
   * The compiler can automatically convert between compatible data types, such as converting an **int** to a **float** when assigning an integer value to a floating-point variable. This conversion is performed implicitly without the need for explicit casting.
2. **Implicit Conversion Caveats**:
   * Implicit conversions may not always produce the desired results, especially when precision or data loss is involved. For example, dividing two integers and assigning the result to a float may result in loss of decimal precision.
3. **Explicit Type Conversion**:
   * When precise control over type conversion is needed, programmers can use explicit casting. This involves using casting operators such as C-style casts or **static\_cast**.
4. **C-style Casts**:
   * C-style casts allow converting between different types but do not provide compile-time checks for validity. They can lead to potential bugs and should be avoided in modern C++ code.
5. **static\_cast**:
   * **static\_cast** is preferred over C-style casts in C++ as it performs compile-time type checking to ensure the validity of the conversion. It provides safer type conversions and helps prevent common programming errors.
6. **reinterpret\_cast**:
   * **reinterpret\_cast** allows casting between unrelated types, even if they are not convertible. It preserves type qualifiers and is useful for low-level type manipulations, but it should be used with caution due to its potential for introducing subtle bugs.
7. **const\_cast**:
   * **const\_cast** is used to add or remove **const** qualifiers from variables. It allows modifying the constness of a variable, but its usage should be limited to situations where it's necessary and appropriate.
8. **Avoiding Type Conversions**:
   * In general, casting between types should be minimized to maintain code clarity and reduce the risk of errors. Whenever possible, use explicit type declarations and avoid relying on implicit conversions.
9. **Compile-Time Operations**:
   * All casting operators, including **static\_cast**, **reinterpret\_cast**, and **const\_cast**, are performed at compile time. This means that type conversions are resolved and validated during the compilation process, rather than at runtime.
10. **dynamic\_cast**:
    * **dynamic\_cast** is another casting operator provided by C++. Unlike other casts, **dynamic\_cast** is primarily used for runtime type checking and type-safe downcasting in polymorphic class hierarchies. It's not discussed in detail in this text but is important in certain scenarios, particularly in object-oriented programming with inheritanceTop of Form

# Type Conversions



**The compiler will implicitly invoke the corresponding parameterized constructor.**

**Since this is an R-value the call will match with the move assignment. So this object will be constructed through its parameterized constructor and then the temporary will be moved by using one of two assignment we created in the class operators, into the object on the left hand side.**

**this expression invokes the parameterized constructor of the Integer class and initializes this object with the value So, this way the primitive type gets converted into the user defined type.**

**Therefore a constructor can be used to convert one type into another, so constructors also take part in type conversion, And they can be invoked explicitly or implicitly.**





* If we try to assign a string to this Integer, the compiler will search for a constructor in the Integer class that accepts a string type. Because we don't have such a constructor, the code will not compile.
* If we want to do this we should provide that constructor



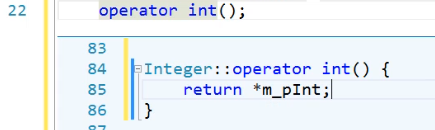
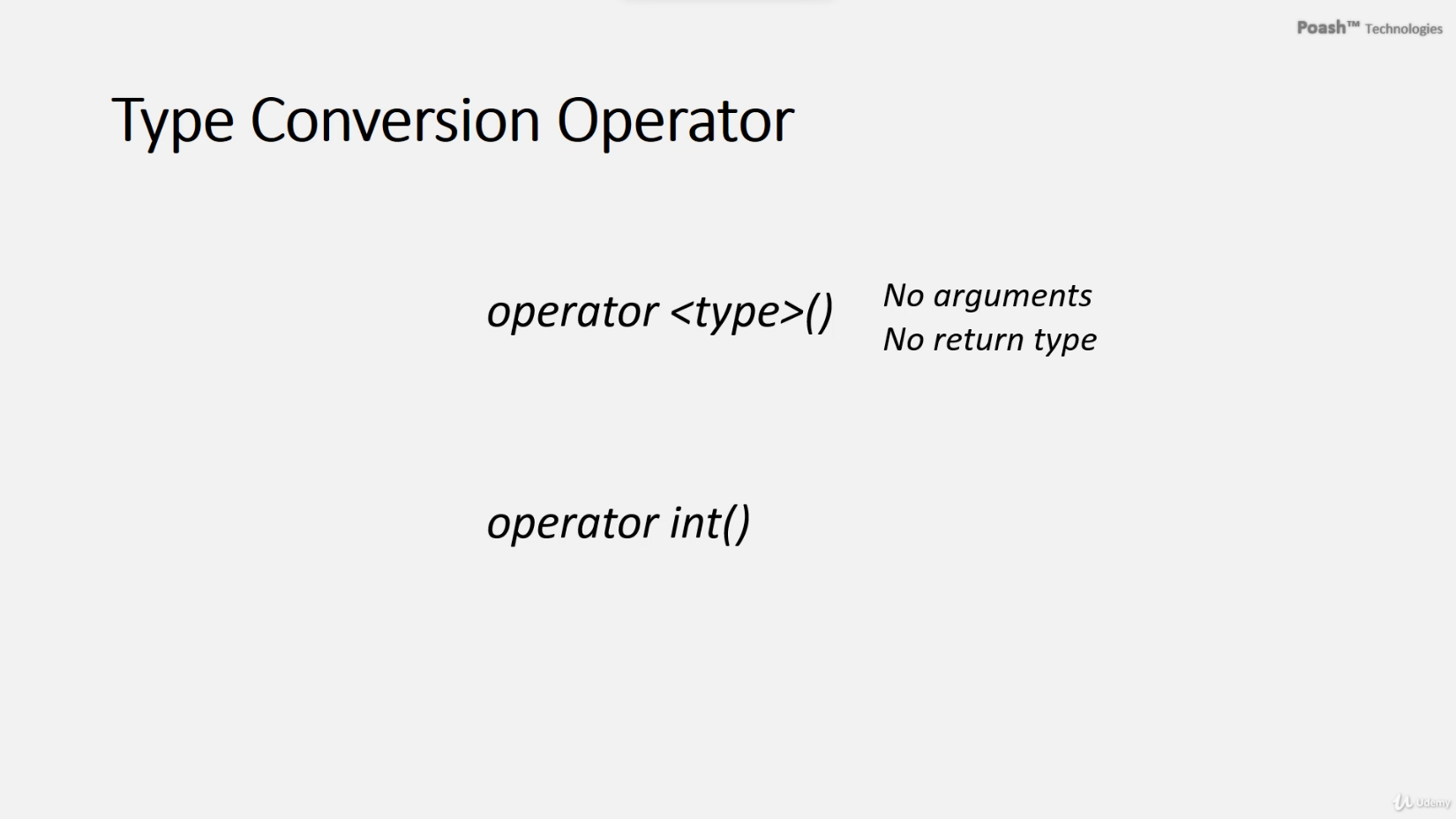
* In some cases implicit casting may be undesirable, we do not want the compiler to automatically use our constructor for implicit conversion.
* Therefore we can mark the constructor with the explicit keyword. Once we do this, The compiler can no longer use that particular constructor for implicit type conversion.
* In most cases, the single argument constructors of your class should be marked with explicit keywords.
* Not all classes may do that because in some cases we have classes that are thin wrappers over primitive types and our Integer class is an example of this type.
* If we do want our users to be able to initialize the user-defined object of Integer with the primitive types. So we may omit this explicit keyword for our Integer class.

type conversion from a primitive type to a user-defined type, particularly focusing on how constructors participate in this conversion process. Here's an explanation:

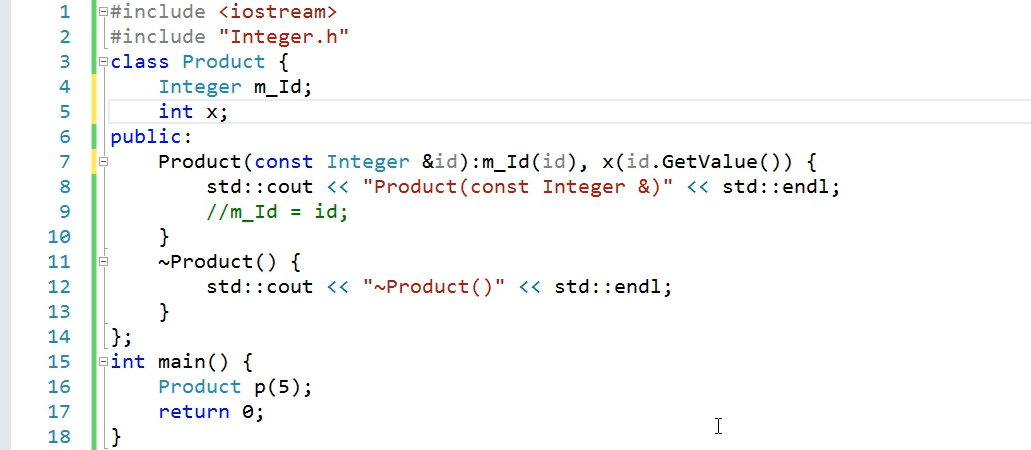
1. **Constructor Invocation for Initialization**:
   * When you initialize a user-defined object with a primitive value, such as **Integer a1 = 5;**, the expression invokes the parameterized constructor of the **Integer** class, passing the primitive value as an argument. This process converts the primitive type (in this case, **int**) into the user-defined type (**Integer**).
2. **Implicit and Explicit Constructor Invocation**:
   * Constructors can be invoked either explicitly or implicitly. In the provided example, the parameterized constructor of the **Integer** class is explicitly invoked. However, in cases like **Integer a2 = 5;**, the compiler implicitly searches for a constructor in the **Integer** class that can accept the primitive type and then invokes it.
3. **Compiler Search for Matching Constructor**:
   * If you try to assign a different type, such as a **string**, to an **Integer** object, the compiler searches for a constructor in the **Integer** class that accepts a **string** argument. If such a constructor is not found, a compilation error occurs.
4. **Constructor Overloading and Type Conversion**:
   * Constructors participate in type conversion by allowing the compiler to convert one type into another. This process is especially useful for initializing user-defined objects with primitive values.
5. **Marking Constructors with explicit Keyword**:
   * In some cases, you may want to prevent the compiler from using constructors for implicit type conversion. You can achieve this by marking the constructors with the **explicit** keyword. This prevents the compiler from automatically using those constructors for implicit conversions.
6. **Considerations for explicit Keyword**:
   * While marking constructors with the **explicit** keyword can prevent implicit type conversions, it may not be suitable for all classes. For example, in thin wrapper classes like the **Integer** class, it's often desirable to allow implicit conversions from primitive types to user-defined types.

Overall, constructors play a crucial role in type conversion from primitive types to user-defined types in C++, allowing for seamless initialization of objects with different types of values.

* So remember constructors are used by the compiler implicitly to convert one type into another.



# Initialization Vs. Assignment & Member Initialization List



**This approach will invoke more functions**

* **Default constructor to create Integer object m\_Id in class**
* **Assignment call due to This line**

**Parameterized constructor convert 5 to Integer object**

**member initializer list**

To convert a user-defined type into a primitive type in C++, you can implement a type conversion operator in the user-defined class. This operator allows the compiler to convert objects of the class into the desired primitive type. Here's how you can do it:

1. **Define the Type Conversion Operator**:
   * Begin by defining the type conversion operator function inside the user-defined class.
   * The operator function starts with the keyword **operator** followed by the type to which the conversion needs to be performed (in this case, **int**).
   * The operator function does not accept any arguments and does not have a return type.
2. **Implement the Operator Function**:
   * Inside the operator function definition, return the value that represents the object's state in the primitive type.
3. **Using the Type Conversion Operator**:
   * Once the type conversion operator is defined in the class, the compiler can use it to implicitly convert objects of the class into the primitive type.
   * If explicit conversion is needed, you can explicitly mention the cast using the type conversion operator.
4. **Considerations**:
   * Using the type conversion operator can cause confusion in some code, especially if implicit conversions occur unexpectedly.
   * To prevent implicit conversions, you can use the **explicit** keyword on the operator function. This ensures that the compiler never uses implicit conversion and requires explicit casting.

Here's a summary of the steps to convert a user-defined type into a primitive type using a type conversion operator in C++:

A screenshot of a computer code

Description automatically generated

In the example above, the **Integer** class defines a type conversion operator that allows objects of the class to be implicitly or explicitly converted to **int**. The compiler invokes this operator when converting **obj** to **int**, either implicitly or explicitly.

Initialization and assignment are two distinct operations in programming, especially in languages like C++.

**Initialization**:

1. **Definition**: Initialization refers to the process of giving an initial value to a variable or an object at the time of its creation.
2. **Syntax**: Initialization occurs when a variable or an object is declared and assigned an initial value using the constructor or an initializer list.
3. **Example**: In C++, initialization can happen using constructors or initializer lists. For example:

A close-up of a white background

Description automatically generated

**Assignment**:

1. **Definition**: Assignment refers to the process of assigning a new value to an existing variable or object after its creation.
2. **Syntax**: Assignment occurs when a variable or an object already exists, and a new value is assigned to it using the assignment operator (**=**).
3. **Example**: In C++, assignment is done using the assignment operator. For example:

A close up of a text

Description automatically generated

**Key Differences**:

1. **Timing**: Initialization happens when a variable or an object is created, while assignment can happen after the creation.
2. **Syntax**: Initialization uses constructors or initializer lists, while assignment uses the assignment operator.
3. **Purpose**: Initialization sets the initial value of a variable or an object, while assignment changes the value of an existing variable or object.

**Preference**:

Initialization is generally preferred over assignment because it:

* Often leads to more efficient code.
* Prevents the creation of temporary objects in some cases.
* Allows for better compiler optimizations.

In C++, using member initializer lists for initialization, especially for class members, is preferred over assignment because it ensures that members are initialized in the correct order and can lead to better performance and readability.