THE STANDARD LIBRARY
THE STANDARD TEMPLATE LIBRARY (STL)

Writing a program from scratch every time would be a tedious task.

Many programs require similar functions, such as reading input from the keyboard, calculating square roots, and sorting data records into specific sequences.

C++ includes a vast amount of pre-existing code that offers various features, saving you the hassle of writing the code from scratch.

Examples are numerical calculations, string processing, sorting and searching,

organizing and managing data, and input and output.

All this standard code is defined in the **Standard Library**.

The Standard Template Library (STL), as a subset of the C++ Standard Library, contains function and class templates for managing and processing data in various ways.

With each new release of the C++ standard, the variety of types and functions also grows.

This chapter does not (cannot) describe the standard library in detail.

It would be best if you referred to books and online documents.

Smart Pointers:

The pointers we have covered up to now are referred to as **raw pointers**.

Variables of raw pointers contain only an address.

They are a part of the C++ language.

A smart pointer is a **class template** that enables the creation of objects that behave like raw pointers.

These objects contain an address and can be utilized in similar ways.

One of the most significant advantages of using a smart pointer is that we do not need to free the memory manually using the delete or delete[] operator.

We create the object and then let the system take care of deleting it at the correct time.

No garbage collector runs in the background (like in Java and C#); memory is managed according to the standard C++ scoping rules so that the runtime environment is faster and more efficient.

There are three types of smart pointers, defined in the `std` namespace:

- `unique_ptr<T>`
- `shared_ptr<T>`
- `weak_ptr<T>`

Smart Pointers (contd):

`unique_ptr<T>`:

It is an object of a template that behaves as a pointer to type `T`.

It is "unique" because there can be only one single `unique_ptr<T>` object (pointer) containing the same address.

In other words, there can never be two or more `unique_ptr<T>` objects simultaneously pointing to the same memory address.

Example:

Unique pointers to `ColoredPoint` objects

```
int main(){
std::unique_ptr<ColoredPoint> ptr1 {new ColoredPoint{10,20,Color::Green }};
{ // A new scope
  auto ptr2{ std::make_unique<ColoredPoint>(30, 40, Color::Blue) };
  ptr2->print();
} // End of scope    // object pointed to by ptr2 is deleted automatically
ptr1->print();
return 0;           // object pointed to by ptr2 is deleted automatically
}
```

See Example e10_1.cpp

Smart Pointers (contd):

`shared_ptr<T>`:

Different than `unique_ptr<T>`, there can be any number of `shared_ptr<T>` objects that contain or share the same address.

Now, we can make a copy of the pointer.

The data pointed to by shared pointers is deleted only if all the pointers holding that memory get out of scope.

This is done by maintaining a **reference counter**.

The reference counter keeps track of how many pointers are pointing to a particular memory location.

The destructor will check the reference counter and free the memory only if the reference counter value is 1.

Example:

```
std::shared_ptr<ColoredPoint> ptr1 {new ColoredPoint{10,20,Color::Green }};
{// A new scope
  std::shared_ptr<ColoredPoint> ptr2{ ptr1 };      // Copy of the   pointer
} // End of scope. The object will not be deleted.
return 0;  // The object is deleted.
```

See Example e10_2.cpp

Smart Pointers (contd):

`weak_ptr<T>`:

The `weak_ptr` is similar to the shared_ptr.

The only difference is that when we create a `weak_ptr` to a `shared_ptr`, the reference count does not increase.

Therefore, the smart pointer will free the memory regardless of whether the `weak_ptr` is still in scope or not.

Example:

```
std::weak_ptr<ColoredPoint> ptr1;
{     // A new scope
 std::shared_ptr<ColoredPoint> ptr2{new ColoredPoint{10,20,Color::Green }};
 ptr1 = ptr2;              // weak_ptr points to same object as shared_ptr
}     // End of scope. The object will be deleted.
// smart_ptr1->print();  The object does not exist
// smart_ptr still exist.
cout << smart_ptr1.use_count() << endl;
// The Number of pointers sharing the same object. weak_ptr does not count
return 0;
}
```

See Example e10_3.cpp

This chapter will be extended.

The Standard Template Library (STL)

- Containers
- Algorithms
- Iterators