C++ Programming I

Repetition

C++ Programming
May 31, 2018

Dr. P. Arnold
Bern University of Applied Sciences

Agenda

- Repetition
- Functions
  - Passing Data to Functions
- Pointers, References, Arrays and Dynamic Memory
- OOP - Classes and Objects
- constructors
- Inheritance
- Polymorphism
- Operators & Casts
- Templates
- C++11 Smart Pointers
- STL

Repetition

Functions

Functions

Prototype, Definition, Overloading

```cpp
// Prototypes
double area(double radius); // for circle
double area(double radius, double height); // overloaded cylinder


// Definition for circle
double area(double radius)
{
   return Pi * radius * radius;
}

// Definition Overloaded for cylinder
double area(double radius, double height)
{
   // reuse the area of circle
   return 2 * area (radius) + 2 * Pi * radius * height;
}
```

- The the compiler determines the most appropriate definition to use by comparing the argument types you have used to call the function
- The process of selecting the most appropriate overloaded function is called **overload resolution or signature matching**

Passing Data to Functions

In C++ there are three different ways to pass data to a function.
Passing:

1. by value:
   `void passByValue(int value);`
2. by reference:
   `void passByReference(int& valueRef);`
3. by pointer:
   `void passByPointer(int* valuePtr);`

- All have different characteristics when it comes to efficiency, storage and behaviour
- We'll focus on 1 & 2
- Passing by pointer is a legacy method used by C-style programs (or function pointers)

Passing by Value

Passing a Copy

```cpp
#include <iostream>
using namespace std;

int square(int x);

int main()
{
    int x = 2;

    cout << "The square of " << x << " is "
         << square(x) << endl;

    return 0;
}

int square(int x)
{
    return x * x;
}
```

- The underlying object is copied using its copy constructor
- Additional memory allocated
- Function works on the copy only!
- For large objects there will be a performance impact

Passing by Reference

Reference

```cpp
#include <iostream>
using namespace std;

int square(int& x);

int main()
{
    int x = 2;

    cout << x << "^2 is " << square(x) << endl;

    cout << x << "^2 is " << square(x) << endl;

    return 0;
}

int square(int& x)
{
   return  x *= x;
}
```

- Underlying object not copied
- The function is given the memory address of the object itself
- Original object can be modified! **Possibility of bugs!**

Passing by Reference to Const

Const Reference

```cpp
#include <iostream>
using namespace std;

int square(const int& x);

int main()
{
    int x = 2;

    cout << "The square of " << x << " is "
         << square(x) << endl;

    return 0;
}

int square(const int& x)
{
    //x *= x; // compilation error! x-cant be changed
    return x * x;
}
```

- No copy AND no modification
- Interface is precise about its intent
- Efficient and safe

Use Reference

Fetching the Result of a function as Reference Parameter

Result as Reference Parameter

```
#include <iostream>
using namespace std;

void square(const int& x, int& result);

int main()
{
    int x = 2;
    int result = 0;

    square(x, result);
    cout << "The square of " << x << " is "
         << result << endl;

    return 0;
}

void square(const int& x, int& result)
{
    result =  x * x;
}
```

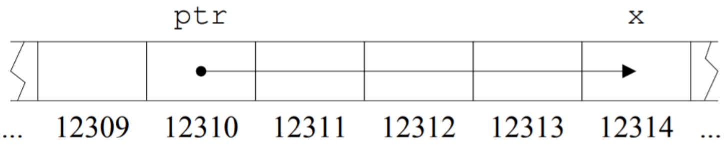

Bern University of Applied Sciences

Pointers, References, Arrays and Dynamic Memory

Variables and Memory

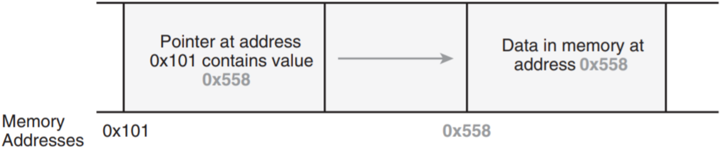
Background I

- When you define a variable, the computer associates the variable name with a particular location in memory and stores a value there. When you refer to the variable by name in your code, the computer must take two steps:
  1. Look up the address that the variable name corresponds to
  2. Go to that location in memory and retrieve or set the value it contains
- C++ allows us to perform either one of these steps independently on a variable with the `&` and `*` operators:
  1. `&x` evaluates to the address of `x` in memory.
  2. `*( &x )` takes the address of `x` and dereferences it - it retrieves the value at that location in memory. `*( &x )` thus evaluates to the same thing as `x`



Variables and Memory
Background II

- A pointer is also a variable storing an **address** in memory. Just the same way as a variable of type `int` is used to contain an integer value, a pointer variable is used to contain a memory address



- A pointer occupies space in memory, e.g. `0x101`, equal to a `int` variable
- The value contained in a pointer,e.g.`0x558` is interpreted as a memory address
- A pointer is a special variable that points to a location in memory
- A pointer that stores the address of some variable `x` is said to **point to** `x`

Pointers
Const Pointers

- There are two places the `const` keyword can be placed within a pointer variable declaration. This is because there are two different variables whose values you might want to forbid changing: the pointer itself and the value it points to.

1. `const int * ptr;`
   Declares a changeable pointer to a **constant integer**. The integer value cannot be changed through this pointer, but the pointer may be changed to point to a different constant integer.
2. `int * const ptr;`
   Declares a **constant pointer** to changeable integer data. The integer value can be changed through this pointer, but the pointer may not be changed to point to a different constant integer.
3. `const int * const ptr;`
   Declares a **constant pointer** to a **const integer**

Memory Allocation

C++ supports three basic types of memory allocation:

1. **Static memory** allocation happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. **Automatic memory** allocation happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited
3. **Dynamic memory** allocation is done by the programmer!

- Both static and automatic allocation have two things in common:
  1. The size of the variable / array must be known at compile time
  2. Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

Memory Allocation

C++ supports three basic types of memory allocation:

1. **Static memory** allocation happens for static and global variables. Memory for these types of variables is allocated once when your program is run and persists throughout the life of your program.
2. **Automatic memory** allocation happens for function parameters and local variables. Memory for these types of variables is allocated when the relevant block is entered, and freed when the block is exited
3. **Dynamic memory** allocation is done by the programmer!

- Both static and automatic allocation have two things in common:
  1. The size of the variable / array must be known at compile time
  2. Memory allocation and deallocation happens automatically (when the variable is instantiated / destroyed).

```cpp
int globalVar = 100; // static memory

int main()
{
    if(true)
    {
        int autoVar = 23; // automatic memory
    } // autoVar freed

    int* dynArray = new int[100]; // dynamic array
    delete [] dynArray; // free manually
    return 0;
} // globalVar freed
```

Dynamic Memory Allocation

Keywords `new` and `delete`

- C++ supplies you two operators, `new` and `delete`, for the management of the memory consumption of your application

```
Type* ptr = new Type; // allocate memory
delete ptr; // release memory allocated above

Type* ptr = new Type[numElements]; // allocate a block
delete[] ptr; // release block allocated above
```

- Allocate objects with `new` **and** free with `delete`
- When declaring an array, the array **is** a pointer to the first element!
- Free arrays with `delete[]`

C++ is not C

Use `new` and `delete` and not C-style `malloc` and `free`

References

References vs Pointers

- When we write `void f(int &x)...` and call `f(y)`, the reference variable `x` becomes another name - an alias - for the value of `y` in memory. We can declare a reference variable locally, as well:

```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

References

References vs Pointers

- When we write `void f(int &x)...` and call `f(y)`, the reference variable `x` becomes another name - an alias - for the value of `y` in memory. We can declare a reference variable locally, as well:

```
int y;
int &x = y; // Makes x a reference to, or alias of, y
```

- References are similar to pointers but are dereferenced every time they are used. The only differences between using pointers and using references are:
  - References are sort of **pre-dereferenced** – you do not dereference them explicitly
  - You cannot change the location to which a reference points, whereas you can change the location to which a pointer points. Because of this, **references must always be initialized** when they are declared
  - `nullptr` references are not possible

Pointers and References

Different Faces of * and &

- The usage of the * and & operators with pointers/references can be confusing. The * operator is used in two different ways:
  1. When **declaring a pointer**, * is placed before the variable name to indicate that the variable being declared is a pointer and not a value
  2. When using a pointer that has been set to point to some value, * is placed before the pointer name to **dereference** it - to access or set the value it points to (**indirection operator**)

- A similar distinction exists for &, which can be used either:
  1. to indicate a **reference data type** (as in `int &x;`), or
  2. to take the **address of** a variable (as in `int *ptr = &x;`).

Pointers and Arrays

Arrays are Pointers, that means:

1. When declaring an array `int a[n]`, the array, i.e. `a` is of type `int* a` and the value of `a` is the address of the first element
2. When dereferencing the array with `*a` one gets the value of the first element.

- `a` is a pointer and not the full array - `a[0]` and `*a` are equivalent

```
int* a = new int[n]; // a is adress of first element

int firstValue = *a; // is the value of the first element

// Number elements is unknown!
int aSize = sizeof(a)/sizeof(*a); // aSize = 1!
// Prefer vector or save size
```

Pointers and Arrays

Array-Pointer Dualism

```
// Static Allocation - Size set at compile time
// ------------------------------------------------------------

int x[10] = {0};  // 10 elements of type int initialized to 0
int first = x[0]; // indexing first element
int last = x[9];  // indexing last element

int x[n]; // Compile error - size must be known

// Dynamic Allocation - Size set at runtime
// ------------------------------------------------------------

int* x = new int[n];

// Access - Arrays are pointers!
// ------------------------------------------------------------

int x = *(a+index); // pointer access is the same as
int x = a[index];   // index access
```

- The name of an array is actually a pointer to the first element in the array
- Array indices start at 0: the first element of an array is the element that is 0 away from the start of the array

OOP - Classes and Objects

Constructors

```cpp
#include <iostream>
#include <string>
using namespace std;

class Human
{
private:
    string m_name;
    int m_age;

public:
    Human(string name = "Adam", int age = 25)
        :m_name(name), m_age(age)
    {
        cout << "Constructed a human called " << m_name;
        cout << ", " << m_age << " years old" << endl;
    }
};
```

- Destructor
- Copy Constructor
- Move Constructor

Note!

The compiler generates default constructors and operators for you!

constructors

Constructors

```cpp
class Demo
{
public:
    Demo();                         // Constructor
    ~Demo();                        // Destructor
    Demo(const Demo& copySrc);      // Copy constructor
    Demo(Demo&& copySrc);           // Move constructor

};
```

- The compiler can be forced to create a constructor
  `Demo() = default;`
- The compiler can be forced to delete a constructor
  `Demo() = delete;`

Note!

In the best case youn don't have to provide any constructor! Let the compiler do the work!

Inheritance

Basic Inheritance

Overriding Base Class's Methods

```cpp
#include <iostream>
using namespace std;

class Fish
{
private:
    bool m_isFreshWaterFish; // not accessible by
                             // derived class
public:
    // Fish constructor
    Fish(bool isFreshWater) : m_isFreshWaterFish(isFreshWater){}

    void swim() // base class method
    {
        if (m_isFreshWaterFish)
            cout << "Swims in lake" << endl;
        else
            cout << "Swims in sea" << endl;
    }
};
```

- Make members private and use initialization lists and overloaded constructors of base class

Basic Inheritance

Overriding Base Class’s Methods

- If the derived classes implements the same methods with the same signatures as in the base class it inherits from, it **overrides** those methods

```
class Tuna : public Fish
{
public:
    Tuna() : Fish(false) {}

    void swim() // Overriding base class method
    {
        cout << "Tuna swims fast" << endl;
    }
};

class Carp : public Fish
{
public:
    Carp() : Fish(true) {}

    void swim() // Overriding base class method
    {
        cout << "Carp swims slow" << endl;
    }
};
```

Basic Inheritance

Overriding Base Class's Methods

```
int main()
{
    Carp carpFish;
    Tuna tunaFish;

    carpFish.swim(); // -> Carp swims slow
    tunaFish.swim(); // -> Tuna swims fast

    return 0;
}
```

- The method `swim()` of the appropriate base class is called
- The only way to invoke `Fish::Swim()` is by having main() use the scope resolution operator `(::)` in explicitly invoking `Fish::Swim()`

```
Tuna tunaFish;

tunaFish.swim();        // will invoke Tuna::swim()

tunaFish.Fish::swim(); // invokes Fish::swim()
                       // using instance of Tuna
```

Basic Inheritance

Order of Construction & Destruction

```cpp
#include <iostream>
using namespace std;

class FishMember
{
public:
   FishMember(){cout << "FishMember constructor" << endl;}
   ~FishMember(){cout << "FishMember destructor" << endl;}
};

class Fish // is base class
{
protected:
   FishMember m_fishMember; // composition with FishMember class

public:
   // Fish constructor
   Fish(){cout << "Fish constructor" << endl;}
   ~Fish(){cout << "Fish destructor" << endl;}
};
```

- This example show the order of construction and destruction when **inheritance and composition** is involved

Basic Inheritance

```
class TunaMember // member of derived
{
public:
    TunaMember(){cout << "TunaMember constructor" << endl;}
    ~TunaMember(){cout << "TunaMember destructor" << endl;}
};

class Tuna: public Fish // derives from base
{
private:
    TunaMember m_tuneMember;

public:
    Tuna(){cout << "Tuna constructor" << endl;}
    ~Tuna(){cout << "Tuna destructor" << endl;}
};

int main()
{
    Tuna tuna;
}

//  FishMember constructor
//  Fish constructor          --> base class finished
//  TunaMember constructor
//  Tuna constructor          --> dervice class finished
//  Tuna destructor
//  TunaMember destructor     --> derived class destructed
//  Fish destructor
//  FishMember destructor     --> base class destructed
```

Basic Inheritance

Access Levels

- So far we have always used the most common access specifier *public* to to derive from base class, thus called *public*-inheritance
- Recap **Access Levels of components:**
  1. **public:** accessible everywhere
  2. **private:** accessible only in methods of the own class
  3. **protected:** accessible only in methods of the own class or derived class
- Using inheritance the **access levels** or visibility of the derived components in the derived classes can be changed:
  1. `class A: public B`
     access level not changed
  2. `class A: protected B`
     access level changed from `public` to `protected`
  3. `class A: private B`
     access level `public` and `protected` changed to `private`

```
class Base
{
    // ... base class members and methods
};
class Derived : private Base // or protected Base
        // private inheritance
{
    // ... derived class members and methods
};
```

Basic Inheritance
Access Specifier

- The following table summarizes the possible access level modifications

Access Specifier In base class	Access Specifier when inherited publicly	Access Specifier when inherited privately	Access Specifier when inherited protectedly
Public	Public	Private	Protected
Private	Inaccessible	Inaccessible	Inaccessible
Protected	Protected	Private	Protected

- `private` and `protected` inheritance describe a **has-a** relationship
- For simplicity, prefer composition over `private` and `protected` inheritance!

Polymorphism

Polymorphic Behavior

The keyword `virtual`

```
class Animal
{
protected:
    std::string m_name;
    Animal(std::string name) : m_name(name) {}

public:
    std::string getName() { return m_name; }
    virtual std::string speak() { return "???"; }
};

class Cat: public Animal
{
public:
    Cat(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Meow"; } // virtual
};

class Dog: public Animal
{
public:
    Dog(std::string name) : Animal(name) {}
    virtual std::string speak() { return "Woof"; } // virtual
};

void report(Animal &animal)
{
    std::cout << animal.getName() << " says " << animal.speak();
}
```

Polymorphic Behavior

The keyword `virtual`

```cpp
int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

- Output?

Polymorphic Behavior

The keyword `virtual`

```
int main()
{
    Cat cat("Fred");
    Dog dog("Garbo");

    report(cat);
    report(dog);
}
```

- Output?

```
Output:

Fred says Meow
Garbo says Woof
```

- It works!

Polymorphic Behavior

The keyword `virtual`

- Similarly, the following example works

```
Cat fred("Fred"), misty("Misty"), zeke("Zeke");
Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

// Set up an array of pointers to animals, and set those
// pointers to our Cat and Dog objects
Animal *animals[] = { &fred, &garbo, &misty, &pooky, &truffle,
     &zeke };
for (int i=0; i < 6; ++i)
{
    cout << animals[i]->getName() << " says " <<
        animals[i]->speak() << endl;
}
```

- Produces the output:

Polymorphic Behavior

The keyword `virtual`

- Similarly, the following example works

```
Cat fred("Fred"), misty("Misty"), zeke("Zeke");
Dog garbo("Garbo"), pooky("Pooky"), truffle("Truffle");

// Set up an array of pointers to animals, and set those
// pointers to our Cat and Dog objects
Animal *animals[] = { &fred, &garbo, &misty, &pooky, &truffle,
     &zeke };
for (int i=0; i < 6; ++i)
{
    cout << animals[i]->getName() << " says " <<
         animals[i]->speak() << endl;
}
```

- Produces the output:

```
Output:

Fred says Meow
Garbo says Woof
Misty says Meow
Pooky says Woof
Truffle says Woof
Zeke says Meow
```

- The signature of the derived class function must exactly match the signature of the base class virtual function

Polymorphic Behavior

override Specifier

```
class A
{
public:
    virtual const std::string getName1(int x) { return "A"; }
    virtual const std::string getName2(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const std::string getName1(short int x)
    { // note: parameter is a short int
        return "B";
    }
    virtual const std::string getName2(int x) const
    { // note: function is const
        return "B";
    }
};

int main()
{
    B b;
    A &BaseRef = b;
    std::cout << BaseRef.getName1(1) << std::endl; // -> A
    std::cout << BaseRef.getName2(2) << std::endl; // -> A

    return 0;
}
```

Polymorphic Behavior

`override` Specifier

```cpp
class A
{
public:
    virtual const std::string getName1(int x) { return "A"; }
    virtual const std::string getName2(int x) { return "A"; }
    virtual const std::string getName3(int x) { return "A"; }
};

class B : public A
{
public:
    virtual const std::string getName1(short int x) override {
        return "B"; } // compile error, not an override

    virtual const std::string getName2(int x) const override {
        return "B"; } // compile error, not an override

    virtual const std::string getName3(int x) override {
        return "B";  } // okay,  is an override
};
```

- There is no performance penalty for using the override specifier.

Tipp

Apply the override specifier to every intended override function you write.

Polymorphic Behavior

`final` Specifier

- Use the `final` specifier to prohibit overriding a function

```cpp
class A
{
public:
    virtual const std::string getName() { return "A"; }
};

class B : public A
{
public:
    // final specifier makes this function no longer overridable
    virtual const std::string getName() override final
    {
        return "B";
    } // okay, overrides A::getName()
};

class C : public B
{
public:
    virtual const std::string getName() override
    {
        return "C";
    } // compile error: overrides B::getName(), which is final
};
```

Virtual Destructors

Why we need Virtual Destructors?

- Similarly, as for other virtual function, calling a function using a pointer of type `Base*` that actually points to `derived*` will call the Base's class function if not marked as `virtual`
- The same holds for the destructor
- Check next example

Virtual Destructors

Why we need Virtual Destructors?

```cpp
#include <iostream>
class Base
{
public:
    ~Base() // note: not virtual
    {
        std::cout << "Calling ~Base()" << std::endl;
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
    {
        m_array = new int[length];
    }

    ~Derived() // note: not virtual
    {
        std::cout << "Calling ~Derived()" << std::endl;
        delete[] m_array;
    }
};
```

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived ;
    delete base;

    return 0;
}
```

- Output? What constructors and destructors are called?

Virtual Destructors

Why we need Virtual Destructors?

```cpp
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived ;
    delete base;

    return 0;
}
```

- Output? What constructors and destructors are called?

```
Output:

Calling ~Base()
```

- Hoppla! Again a memory leak!

Virtual Destructors

Why we need Virtual Destructors?

```
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()" << std::endl;
    }
};

class Derived: public Base
{
private:
    int* m_array;

public:
    Derived(int length)
    {
        m_array = new int[length];
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()" << std::endl;
        delete[] m_array;
    }
};
```

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived;
    delete base;

    return 0;
}
```

- Output?

Virtual Destructors

Why we need Virtual Destructors?

```
int main()
{
    Derived *derived = new Derived(5);
    Base *base = derived;
    delete base;

    return 0;
}
```

- Output?

```
Output:

Calling ~Derived()
Calling ~Base()
```

- It works!

Operators & Casts

Unary Operators
Declaration of Unary Operators

- As the name suggests, operators that function on a single operand are called `unary operators`
- As a static function or implemented in the global namespace, the structure is given by:

```
return_type operator operator_type (parameter_type)
{
    // ... implementation
}
```

- As a class member, the structure is missing in parameters, because the single parameter that it works upon is the instance of the class itself (`*this`):

```
return_type operator operator_type ()
{
// ... implementation
}
```

Binary Operators

Declaration of Binary Operators

- Operators that function on two operands are called `binary operators`
- The definition of a binary operator implemented as a global function or a static member function is the following:

```
return_type operator operator_type (parameter1, parameter2)
{
    // ... implementation
}
```

- Since one parameter is given by class instance itself (`*this`) definition of a binary operator implemented as a class member is:

```
return_type operator operator_type (parameter)
{
    // ... implementation
}
```

Binary Operators

Copy & Move Assignment =

- Copy Assignment:

```
ClassType& operator=(const ClassType& copySource)
{
    if(this != &copySource) // protection against copy into self
    {
        // copy assignment operator implementation
    }
    return *this;
}
```

- Move Assignment:

```
ClassType& operator=(ClassType&& moveSource)
{
    // check if valid
    if(this != &moveSource && moveSource.m_Ptr != nullptr)
    {
        // move assignment operator implementation
    }
    return *this;
}
```

Note:

Only provide own versions if your class encapsulates raw pointers!

Casting Operators

supplied by C++

- C++ supplies four casting operators:
  1. `static_cast`
  2. `dynamic_cast`
  3. `reinterpret_cast`
  4. `const_cast`

Casting Operators

supplied by C++

- C++ supplies four casting operators:
  1. `static_cast`
  2. `dynamic_cast`
  3. `reinterpret_cast`
  4. `const_cast`
- Explicit & implicit casting
- Up & down casting
- Run-time casting (RTTI)

Templates

Template Functions

Using Template Syntax

- A template function is capable of adapting itself to suit parameters of different types.
- Lets implement the `MAX` macro with templates:

```
template <typename T>
const T& getMax(const T& value1, const T& value2)
{
    return (value1 > value2) ? value1 : value2; // Ternary op.
}
```

Template Functions

Using Template Syntax

- A template function is capable of adapting itself to suit parameters of different types.
- Lets implement the `MAX` macro with templates:

```cpp
template <typename T>
const T& getMax(const T& value1, const T& value2)
{
    return (value1 > value2) ? value1 : value2; // Ternary op.
}
```

- The function can then be called like this:

```cpp
int num1 = 25;
int num2 = 40;
int maxVal = getMax<int>(num1, num2);
int maxVal = getMax(num1, num2);


double double1 = 1.1;
double double2 = 1.001;
double maxVal = getMax<double>(double1, double2);
double maxVal = getMax(double1, double2);
```

Templates

The `<int>` used in the call `getMax` defines the template parameter T, but is not necessary for template functions

Template Classes

Syntax

- Template classes are the templatised versions of C++ classes!
- For example `std::vector<int>` is a class holding integers as type

Template Classes

Syntax

- Template classes are the templatised versions of C++ classes!
- For example `std::vector<int>` is a class holding integers as type
- A simple template class that uses a single parameter T to hold a member variable can be written as the following:

```
template <typename T>
class TemplateClass
{
private:
    T value;
public:
    void setValue(const T& newValue) { value = newValue; }
    T& getValue() { return value; }
};
```

- The type `T` of variable `value` is *instantiated* when the template is used

C++11 Smart Pointers

Smart Pointers

Existing Smart Pointers: `#include <memory>`

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- `std::auto_ptr`

Smart Pointers

Existing Smart Pointers: `#include <memory>`

- `std::unique_ptr`
- `std::shared_ptr`
- `std::weak_ptr`
- ~~`std::auto_ptr`~~ (depriciated since C++ 11, removed in C++ 17)

Garbage Collection

Using Smart pointers is the simplest garbage collector we could think of!

STL

STL - Containers

Sequential Containers

Sequential containers are characterized by a **fast insertion time**, but are relatively **slow in find operations**.

- `std::vector` - Operates like a dynamic array and grows only at the end
- `std::deque` - Similar to `std::vector` except that it allows for new elements to be inserted or removed at the beginning, too
- `std::list` - Operates like a double linked list. Like a chain where an object is a link in the chain. You can add or remove links, *i.e.* objects at any position
- `std::forward_list` Similar to a `std::list` except that it is a singly linked list of elements that allows you to iterate only in one direction

STL - Containers
Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

STL - Containers
Associative Containers

Associative containers store data in a sorted fashion. This results in **slower insertion times**, but **optimized search performance**

- `std::set` - Stores unique values sorted on insertion in a container featuring logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_set` - Stores unique values sorted on insertion in a container featuring near constant complexity $\mathcal{O}(1)$. Available starting C++ 11
- `std::map` - Stores key-value pairs sorted by their unique keys in a container with logarithmic complexity $\mathcal{O}(\log n)$
- `std::unordered_map` - Stores key-value pairs sorted by their unique keys in a container with near constant complexity $\mathcal{O}(1)$(since C++ 11)
- `std::multiset` - Like `set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(\log n)$
- `std::unordered_multiset` - Like `unordered_set`. Additionally, supports the ability to store multiple items having the same value; that is, the value doesn't need to be unique $\mathcal{O}(1)$(since C++ 11)
- `std::multimap` - Like `map` . Additionally, supports the ability to store key-value pairs where keys don't need to be unique. $\mathcal{O}(\log n)$
- `std::unordered_multimap` - Like `unordered_map`. Additionally, supports the ability to store key-value pairs where keys don't need to be unique $\mathcal{O}(1)$ (since C++ 11)

STL - Containers
Container Adapters

Container adapters are variants of sequential and associative containers that have limited functionality and are intended to fulfill a particular purpose

- `std::stack` - Stores elements in a LIFO (last-in-first-out) fashion, allowing elements to be inserted (pushed) and removed (popped) at the top
- `std::queue` - Stores elements in FIFO (first-in-first-out) fashion, allowing the first element to be removed in the order they're inserted
- `std::priority_queue` - Stores elements in a sorted order, such that the one whose value is evaluated to be the highest is always first in the queue

Thank You

Questions

???