CS100 Recitation 10

Contents

Part 1: Overloading

- **Overloading** occurs when two or more *functions* or *operators* share the same name but differ in the **number** or **types** of their parameters.
- **Overloading** enhances *data abstraction* and enables **compile-time polymorphism**.
  - **Data abstraction:** the process of *hiding implementation details* and exposing only the **essential interface**.

Part 1: Overloading

- Function Overloading
  - Usage
  - Overload Resolution
  - Default Parameters & Function Overloading
- Operator Overloading

Function Overloading (*Compile-Time Polymorphism / Static Polymorphism*)

- When a function serve similar purposes in multiple contexts, defining separate functions with different names can clutter the namespace. Instead, we **overload** the function and let the compiler select the appropriate definition based on the *call-site context*.
- Valid overloaded function must have a **unique parameter list**; differences in return type alone do **not** constitute a valid overload.

Usage: General Functions

Suppose we need to implement a program to **calculate the areas of different shapes**.

First, to compute the area of a **circle** given its *radius*:

```
double areaCircle(double radius) {
  return M_PI * radius * radius;
}
```

Additionally, to compute the area of a **rectangle** given its *width* and *height*:

```
double areaRectangle(double width, double height) {
  return width * height;
}
```

Usage: General Functions

```
double areaCircle(double radius);
double areaRectangle(double width, double height);
```

To invoke these functions:

```
double r = 5.0;
double w = 4.0, h = 6.0;

std::cout << "Circle area (r=" << r << "): " << areaCircle(r) << '\n';
std::cout << "Rectangle area (" << w << "×" << h << "): "
  << areaRectangle(w, h) << '\n';
```

Usage: General Functions

However, with **function overloading**, we may unify the names:

```
double area(double radius) { return M_PI * radius * radius; }

double area(double width, double height) { return width * height; }
```

And then use them as if they were the **same function**:

```
double r = 5.0;
double w = 4.0, h = 6.0;

std::cout << "Circle area (r=" << r << "): " << area(r) << '\n';
std::cout << "Rectangle area (" << w << "×" << h << "): "
  << area(w, h) << '\n';
```

Usage: `const`, non-`const` overloading

Usage: `const`, non-`const` overloading

We will examine the evolution of **`Dynarray::at`** to illustrate the use of `const` and non-`const` overloads.

1. **Initial Implementation**

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;

public:
  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

Problem:

The `at` member function can only be called on **non-`const`** instances of `Dynarray`. A **`const`** instance cannot invoke this function because it is not declared as a **`const`** member.

Usage: `const`, non-`const` overloading

2. **Adding a `const` overload**

```
int &Dynarray::at(std::size_t n) const {
  return m_storage[n];
}
```

Although this overload can be called on both `const` and non-`const` objects, it returns a **modifiable reference** even for **`const`** instances, violating **const correctness**.

---

3. **Ensuring safe return type**

```
const int &Dynarray::at(std::size_t n) const {
  return m_storage[n];
}
```

Now, regardless of whether the `Dynarray` instance is **non-`const`**, this overload always returns a **`const`** reference, preventing modification of elements even when the object is mutable.

Usage: `const`, non-`const` overloading

Final Overloaded Implementation

```
class Dynarray {
  std::size_t m_size;
  int *m_storage;

public:
  const int &at(std::size_t n) const {
    return m_storage[n];
  }

  int &at(std::size_t n) {
    return m_storage[n];
  }
};
```

- For `const` objects, only the `const` overload is viable—no overload resolution occurs.
- For **non-**`const` objects, both overloads are viable. However, binding to the `const` overload requires adding `const` to the implicit `this` pointer, whereas the non-`const` overload is an **exact match**, so the compiler selects it.

Usage: `const`, non-`const` overloading

```
const int &Dynarray::at(std::size_t n) const;
int &Dynarray::at(std::size_t n);
```

Note: Member function overloads differ by the implicit `this` parameter:

```
// Conceptual expansions:
const int &Dynarray::at(const Dynarray *this, std::size_t n) { /*...*/ }
int & Dynarray::at(Dynarray *this, std::size_t n) { /*...*/ }
```

Usage: `const`, non-`const` overloading

Recall that unlike `operator[]`, which omits bounds checking, `at()` performs range validation and throws on out-of-range access:

```cpp
class Dynarray {
public:
  const int &at(std::size_t n) const {
    if (n >= m_length)  // `std::size_t` is unsigned
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
  int &at(std::size_t n) {
    if (n >= m_length)
      throw std::out_of_range{"Dynarray subscript out of range."};
    return m_storage[n];
  }
};
```

Reuse Your Code

It is recommended to avoid duplicating logic when overloads share identical implementation. Since both overloads of `at` exhibit the same behavior, we can implement the non-`const` overload by delegating to the `const` version:

- Removing `const` qualification and then adding it back is **riskier** than the reverse. Therefore, the non-`const` overload should call the `const` overload.

```
const int &Dynarray::at(std::size_t n) const;
int &Dynarray::at(std::size_t n) {
return const_cast<int &>(
    static_cast<const Dynarray *>(this)->at(n)
);
}
```

Usage: Constructors

Usage: Constructor Overloadings

(Take the example of the template class `std::vector<T>`)

- **Default Constructor**

```
std::vector<T>();
```

- **Copy Constructor**

```
std::vector<T>(const std::vector<T> &other);
```

- **Move Constructor**

```
std::vector<T>(std::vector<T> &&other);
```

- **Custom Constructor**

```
std::vector<T>(size_t count, const T &value);
```

Usage

```cpp
#include <vector>

int main() {
  // 1. Default constructor
  std::vector<int> v1;

  // 2. Copy constructor
  std::vector<int> v2(v1);

  // 3. Move constructor
  std::vector<int> v3(std::move(v2));

  // 4. Custom constructor
  std::vector<int> v4(5, 42);

  return 0;
}
```

Usage: Conclusion

- There is **no** operational difference between the 3 examples above. **Overloading** shifts complexity to the class designer while simplifying the interface for the user.
- Effective function overload design should maximize:
  - **Semantic similarity** across overloads
  - **Natural syntax** so that users remain unaware they invoke distinct functions

Template (Preview)

When multiple functions share identical logic but differ only by type—and cannot be overloaded due to identical parameter patterns—**templates** provide a solution. Template instantiation can be viewed as a form of compile-time overloading.

```cpp
int addInt(int a, int b) {
  return a + b;
}
double addDouble(double a, double b) {
  return a + b;
}
```

```cpp
int add(int a, int b) {
  return a + b;
}
double add(double a, double b) {
  return a + b;
}
```

```cpp
template <typename T>
T add(T a, T b) {
  return a + b;
}
```

In this example, all three approaches compile to simple binary operations. **Templates** simply allow the compiler to generate as many `add<T>` instantiations as required.

Overload Resolution

The rules to select the only function to finally called among those who can be called

- Functions with one parameters
- Functions with multiple parameters

Overload Resolution: Functions with one parameters

- Identify the set of ***candidate functions***

  > Before overload resolution begins, the functions selected by name lookup and template argument deduction are combined to form the set of candidate functions.

- From the set of ***candidate functions***, identify the set of ***viable functions***

  > If the functions has the identical number of arguments, each parameter can be converted to the corersponding arguments through at least one implicit conversion sequence and etc.

- Analyze the set to determine the single best ***viable functions*** through:
  Exact Match $\Longrightarrow$ Promotion $\Longrightarrow$ Standard type conversion $\Longrightarrow$ User-defined type conversion

Exact Match

- Value-to-rvalue conversion

```
void f(int);

int x = 42; // x is an lvalue
f(x);
// x → rvalue when passed by value
```

- Array-to-pointer conversion

```
void f(int *a);

int ar[10];
f(ar);
```

- Function-to-pointer conversion

```
typedef int (*fp)(int);
void f(int, fp);

int g(int);
f(5, g);
```

- Qualification conversion
  - Converting pointer (only) to `const` pointer

```
void g(const int *);

int a = 5; int *p = &a;
g(p);
```

Overload Resolution: Promotion

Promotion: integral promotion, floating-point promotion

> Floating point promotion: A prvalue of type float can be converted to a prvalue of type double. The value does not change. This conversion is called floating-point promotion.

Typical conversions:

- `char` → `int`; `float` → `double`
- `enum` → `int` / `short` / `unsigned int` / ...
- `bool` → `int`

Overload Resolution: Conversion

Conversion: integral conversion, floating-point conversion, floating-integral conversion, pointer conversion, pointer-to-member conversion, boolean conversion, user-defined conversion of a derived class to its base

- **Integral conversion**

```
void f(short);
unsigned int ui = 1000u;
f(ui);      // ui → short
```

- **Floating-point conversion**

```
void f(float);
double d = 3.14159;
f(d);       // double → float
```

- **Pointer conversion**

```
void f(void*);
char* cp = nullptr;
f(cp);      // char* → void*
```

- **Bool conversion**

```
void f(bool);
int i = -1;
f(i);       // int → bool
```

Example 1: Overload Resolution with one parameter

In the context of a list of function prototypes:

```
int g(double); // F1
void f(); // F2
void f(int); // F3
double h(void); // F4
int g(char, int); // F5
void f(double, double = 3.4); // F6
void h(int, double); // F7
void f(char, char*); // F8
```

Which function will be called `f(5.6)`? List out the candidate functions, viable functions, and the final best viable function.

Example 1: Overload Resolution with one parameter

```
int g(double); // F1
void f(); // F2
void f(int); // F3
double h(void); // F4
int g(char, int); // F5
void f(double, double = 3.4); // F6
void h(int, double); // F7
void f(char, char*); // F8
```

Resolution

1. Candidate functions (by name): F2, F3, F6, F8
2. Viable functions (by # of parameters): F3, F6
3. Best viable function (by parameter type `double` – Exact Match): **F6**

Example 2: Overload Resolution fails

- Consider the overloaded function signatures:

```cpp
int fun(float a) { /*...*/ }              // Function 1
int fun(float a, float b) { /*...*/ }    // Function 2
int fun(float x, int y = 5) { /*...*/ } // Function 3
```

- In `main()`:

```cpp
int main() {
  float p = 4.5, t = 10.5;
  int s = 30;

  fun(p, s);  // CALL 1
  fun(t);     // CALL 2
  return 0;
}
```

- **CALL 1**: Viable functions: Function 2 & Function 3
- **CALL 1**: Best match → Function 3
- **CALL 2**: Viable functions: Function 1 & Function 3
- **CALL 2**: Ambiguous (no single best match)

Overload Resolution with Multiple Parameters

> For overload resolution between functions F1 and F2:
>
> *F1 is better than F2 if, for some argument i, F1 has a better conversion than F2, and for other arguments F1 has a conversion which is not worse than F2.*

- **Example 1** (ambiguous):

```
int fun(int, int, int);             // F1
int fun(double, double, double);    // F2
int main() {
  fun(5, 5, 2.0);
}    // Ambiguous
```

  The above is ambiguous because neither `F1` nor `F2` has a better conversion than the other: For the first and second argument matching, `F1` overtakes `F2`, however for the third, `F2` overtakes `F1`.

Overload Resolution with Multiple Parameters (cont.)

> For overload resolution between functions F1 and F2:
>
> *F1 is better than F2 if, for some argument i, F1 has a better conversion than F2, and for other arguments F1 has a conversion which is not worse than F2.*

- **Example 2** (F1 wins):

  ```
  int fun(int, int, double);        // F1
  int fun(int, double, double);     // F2

  int main() {
    fun(5, 5, 5);   // F1 wins
  }
  ```

  F1 is better than F2 in the second argument and not worse in the other two arguments, therefore `fun(5, 5, 5)` calls `F2`.

Default Parametesr v.s. Function Overload

Default Parameters

```
int f(int a = 1, int b = 2);
```

```
int x = 5, y = 6;
f();         // a = 1, b = 2
f(x);        // a = x = 5, b = 2
f(x, y);     // a = x = 5, b = y = 6
```

- Function `f` has 2 parameters with defaults
- `f` can be called in 3 forms

<u>The two distinct implementations reach the same usage.</u>

Function Overload

```
int f();
int f(int);
int f(int, int);
```

```
int x = 5, y = 6;
f();          // calls int f();
f(x);         // calls int f(int);
f(x, y);      // calls int f(int, int);
```

- Function `f` is overloaded with up to 3 signatures
- `f` can be called in 3 forms

Default Parameter & Function Overload

Compilers deal with default parameters as a special case of function overloading, since syntaxically, we call the functions (with default parameters) multiple times using the same identifier(name) but different argument lists.

- Additionally, ***if resolvable***, overloaded function can also use default parameters.

```cpp
int Area(int a, int b = 10) {
  return a * b;
}

double Area(double c, double d) {
  return c * d;
}
```

```cpp
int x = 10;
double z = 20.5, u = 5.0;

int t = Area(x);  // Binds int Area(int, int = 10)
std::cout << "Area = " << t; // t = 100

// Binds double Area(double, double)
double f = Area(z, u);
std::out << "Area = " << f; // f = 102.5
```

Example: Failed overloading resolution

```
int f();
int f(int = 0);
int f(int, int);

int main() {
  int x = 5, y = 6;

  f();  // error: call to 'f' is ambiguous matches both f() and f(int = 0)
  f(x); // calls int f(int)
  f(x, y);  // calls int f(int, int)

  return 0;
}
```

Operator Overloading (Revisit)

- Copy and Swap
- Operator `->`
- Function calling operator & Functor

Dynarray: Operator Overloading

Copy-Control

```
Dynarray& operator=(const Dynarray& other);
Dynarray& operator=(Dynarray&& other) noexcept;
```

*Rule of Three/Five:

> Define zero or five of them.
> (Since move semantics hasn't introduced until C++11, before C++11 it's the rule of 3.)

If one of the five copy control members has an user-provided definition, the copmiler should not automatically synthesis other members that user are not user-provided (except for destuctors).

Copy-and-Swap Idiom

> **Idiom**: Programming techniques that are widely used among experienced programmers and are proved useful and idiomatic.

`std::swap` - CppReference

```cpp
template<class T>
void swap(T& a, T& b);
```

- Exchanges the given values `a` and `b`.
- `T` must meet the requirements of move-constructible and move-assignable. Contrary to the method of utilizing a temporary to swap:

  ```cpp
  auto tmp = a; a = b; b = tmp;
  ```

  it has higher requirements of move semantics, while the `tmp` way is more universal. (If `T` is not copyable, it can be replaced with `auto tmp = std::move(a)` and so on).

`std::swap` - Exception Safety

> `noexcept`: the typical use is to append to the end of a functoin's declaratoin and mark the function will not throw any exceptions

Since C++11, `noexcept` supports specification, therefore, the declaration of `std::swap` can written as follows:

```
template<class T>
void swap(T& a, T &b) noexcept(
  std::is_nothrow_move_constructible<T>::value &&
  std::is_nothrow_move_assignable<T>::value
);
```

Use `std::swap` to Swap Dynarray

```
class Dynarray {
public:
  friend void swap(Dynarray &a, Dynarray &b) noexcept {
    using std::swap;
    swap(m_storage, other.m_storage);
    swap(m_length, other.m_length);
  }
};
```

- `Dynarray::swap` is marked `noexcept` (with exception safety) since it's given that `Dynarray` can be move-assigned or move-constructed.
- `using std::swap` guarantees calling `swap(..., ...)` matches the best implementation. Since we name the `swap` function with the same name as `std::swap`, when `using std::swap`, calling `swap(..., ...)` would match the best overloaded function if we have defined some. Otherwise, this will call `std::swap` in

Copy-and-Swap Idiom: Assignment operator

Recall the `Dyanrray` 's copy assginment operator:

```
Dynarray& operator=(const Dynarray& other) {
  if (this != &other) {
    int* new_data = new int[other.m_length];
    for (std::size_t i = 0; i < other.m_length; ++i)
      new_data[i] = other.m_storage[i];
    delete[] m_storage;
    m_storage = new_data;
    m_length  = other.m_length;
  }
  return *this;
}
```

In essence, copy assignment can be divided into copy construct the new and destruct the old. We can use `std::swap` to achieve it.

Copy-and-Swap Idiom: Assignment operator

```
class Dynarray {
public:
  Dynarray &operator=(const Dynarray &other) {
    auto tmp = other;
    swap(*this, tmp);
    return *this;
  }
};
```

- It's easy and clean since it reuse the code of copy constructor on copying `other` and destructor on destruct `tmp`. Additionally, we can copy when passing arguments in advance:

```
Dynarry &Dynarray::operator=(Dynarray other) noexcept {
  swap(*this, other);
  return *this;
}
```

Copy-and-Swap Idiom: Assignment operator

```
Dynarry &Dynarray::operator=(Dynarray other) noexcept {
  swap(*this, other);
  return *this;
}
```

Now let's consider what will happens when passing rlvalue to this function:

- If the arugment is lvalue, it will be `copied` to other and copy constructor will be invoked.
- If the argument is rvalue, it will be moved into `other` and copy consrtuctor will be invoked.

Therefore, this assignment operator not only an copy assignment operator, but also a move assignment operator.

Copy-and-Swap Idiom

The **copy-and-swap idiom** is a canonical way to implement `operator=` in C++ that automatically gives you:

- **Strong exception safety** (if copying fails, the left-hand object is unchanged)
- **Implicit self-assignment safety** (swapping a copy of yourself back in does nothing)
- **Minimal code duplication** (you reuse your copy-constructor and `swap`)
- **Unified copy and move assignment** (a single `operator=` via pass-by-value handles both copy and move)

Dynarray: Operator Overloading

Element Access

```
int& operator[](std::size_t idx);
const int& operator[](std::size_t idx) const;
```

Increment / Decrement

```
Dynarray& operator++();    // ++arr
Dynarray operator++(int); // arr++

Dynarray& operator--();    // --arr
Dynarray operator--(int); // arr--
```

Dynarray: Operator Overloading

Arithmetic & Compound-Assignment

```
Dynarray& operator+=(const Dynarray& rhs);
Dynarray& operator-=(const Dynarray& rhs);
Dynarray& operator*=(const Dynarray& rhs);
Dynarray& operator/=(const Dynarray& rhs);

friend Dynarray operator+(const Dynarray& lhs, const Dynarray& rhs);
friend Dynarray operator-(const Dynarray& lhs, const Dynarray& rhs);
friend Dynarray operator*(const Dynarray& lhs, const Dynarray& rhs);
friend Dynarray operator/(const Dynarray& lhs, const Dynarray& rhs);
```

Dynarray: Operator Overloading

Comparison

```
friend bool operator==(const Dynarray& lhs, const Dynarray& rhs);
friend bool operator!=(const Dynarray& lhs, const Dynarray& rhs);
friend bool operator< (const Dynarray& lhs, const Dynarray& rhs);
friend bool operator> (const Dynarray& lhs, const Dynarray& rhs);
friend bool operator<=(const Dynarray& lhs, const Dynarray& rhs);
friend bool operator>=(const Dynarray& lhs, const Dynarray& rhs);
```

Stream I/O

```
friend std::ostream& operator<<(std::ostream& os, const Dynarray& arr);
friend std::istream& operator>>(std::istream& is, Dynarray& arr);
```

Other overloads

Function Object(Functor)

```
ReturnType operator()(ArgTypes... args) const;
```

Pointer-Like Operators

```
int& operator*()  const;
int* operator->() const;
```

Type Conversion

```
operator SomeOtherType() const;
explicit operator bool() const;
```

Function Object (`operator()`)

An function object is any object for which the function call operator is defined. It's one of the **callable objects** (predicates) in C++, another typical callable object is the **lambda expression**.

```
ReturnType operator()(ArgTypes... args) const;
```

- Overload the function call operator makes the corresponding object behave like a function and encapsulate a callable with state.

```
struct Adder {
  int base;
  Adder(int b) : base(b) {}
  int operator()(int x) const { return base + x; }
};
Adder add5(5);
int result = add5(10); // calls add5.operator()(10) → 15
```

Conversion Operators

Allow user-defined types to convert to other types.

```
operator SomeOtherType() const;
explicit operator bool() const;
```

- **Implicit** conversions happen when no `explicit` keyword is used.
- **Explicit** conversions (e.g. `explicit operator bool()`) prevent unwanted conversions but still work in contexts like `if (obj)`.

Pointer-Like Operators

```
T& operator*() const;
T* operator->() const;
```

Since usually we will gurantee that our defined class should be similar to STL, `p->mem` is usually defined identical to `(*p).mem`, we typically define `operator*` and use the following fixed writing of `operator->`:

```
T& operator*() const {
  /* Your implementation */
}

T* operator->() const {
  return std::addressof(this->operator*());
}
```

`Operator->`

It's called "member of pointer" operator. CppReference

- For built-in types, the expression `E1->E2` is exactly equivalent to `(*E1).E2`. Thus its thus a binary operator.
- If a user-defined operator-> is called, operator-> is called again on the resulting value, recursively, until an operator-> is reached that returns a plain pointer. After that, built-in semantics are applied to that pointer. Thus, user-defined operator-> must either return:
    - A raw pointer (e.g. `T*`), or
    - An object (by value or reference) that itself defines `operator->`.

  In this case, technically, `operator->` is binary operator to return according to left hand side object.

`std::addressof(this->operator*())`

```
T* operator->() const {
  return std::addressof(this->operator*());
}
```

1. Since `this` is a raw pointer, operator-> behaves as `(*).`. `this->operator*()` calls your class's `operator*()`, returning a `T&`—no `operator->` is involved at this step.
2. `std::addressof(...)` yields the actual address of that `T&`, giving you a raw `T*` (**it bypasses any overloaded `operator&`**).
3. Because the return type is `T*`, the next `->` is the built-in pointer arrow, not your overload—so the recursion chain ends.

Part 2 Smart Pointers and RAII Idiom

RAII Idiom

> RAII: Resouce Allocation Is Acquistion

Resource

- A resource in C++ is a facility or concept you **acquire** by executing a statement or expression.
- You later **release** or **dispose** of that resource with a corresponding statement.

Resource	Acquisition	Disposal
Memory	`p = new T;`	`delete p`
Files	`fp = fopen("filename", "r");`	`fclose(fp);`
*Threads	`pthread_create(&p. NULLL, fn. NULL);`	`pthread_join(p, &retVal);`
…	…	…

Resource Usage Issues

- **Leak:** Forgetting to dispose of a resource.
  - Memory: swapping, out of memory, killed by manager
  - Open too many files without closing: run out of file desciptors in the process and cannot open any more files
  - etc.
- **Use-after-disposal:** Using a resource after it’s been released.
- **Double-disposal:** Releasing the same resource twice.

Question: Is there any memory leak?

```
bool processData(SomeDataSource& src) {
  const std::size_t kSize = 1024;
  int* buffer = new int[kSize];

  if (!src.read(buffer, kSize))
      return false;

  displayBuffer(buffer, kSize);
  delete[] buffer;
  return true;
}
```

The above is a function read data `src` to a buffer and then display it. Is there any memory leak? If there is, how to fix it?

Question: Is there any memory leak?

```cpp
bool processData(SomeDataSource& src) {
  const std::size_t kSize = 1024;
  int* buffer = new int[kSize];

  if (!src.read(buffer, kSize)) {
    delete[] buffer;          // cleanup on this path
    return false;
  }

  displayBuffer(buffer, kSize);
  delete[] buffer;              // cleanup on success
  return true;
}
```

How about now? Is there still any memory leak?

Question: Is there any memory leak?

```
bool processData(SomeDataSource& src) {
  const std::size_t kSize = 1024;
  int* buffer = new int[kSize];
  try {
    if (!src.read(buffer, kSize)) {
        delete[] buffer;
        return false;
    }
    displayBuffer(buffer, kSize);
  }
  catch (...) { delete[] buffer; throw; }
  delete[] buffer;                    // cleanup on normal exit
  return true;
}
```

We should also consider the circumstances of excetions.

- It's ugly and annoying even in such a small snippets.

Resource Allocation Is Initialization

Objects have a defined beginning of life, and end of life. Both of those events have code which will automatically run: namely, constructors and destructors. RAII leverages these automatic calls to manage resources reliably.

```cpp
#include <memory>
bool processData(SomeDataSource& src) {
  const std::size_t kSize = 1024;
  std::unique_ptr<int[]> buffer(new int[kSize]);

  if (!src.read(buffer.get(), kSize))
      return false;            // buffer auto-deleted

  displayBuffer(buffer.get(), kSize);
  return true;                 // buffer auto-deleted
}
```

Additonally, you can furtherly ease it by using STL container (E.g., `std::vector`).

RAII

In the pureset sense of the term, this is the idiom where the resource acquistion is done in the constructor of an "RAII class". and resource disposal is done in the destructor of an "RAII class".

- **RAII object**: An object whose class design adheres to the RAII principles.
- **RAII class**: A class embodies RAII concept by tying the resource’s lifetime to the lifespan of an object. Upon the construction of such an object, the necessary resource is acquired, and when the object is destroyed (for example, when it goes out of scope), its destructor automatically releases the resource.
- When programming in the RAII style, the lifetime of a <u>***resource***</u> is bound to the lifespan of the object <u>***instance***</u>.

Ownership

Ownership is the responsibility for managing a resource's lifetime - allocation, use, eventual deallocation.

- Effective resource management requires a well-defined system of accountability. When a resource is transferred to a RAII class, the class assumes full responsibility for its lifecycle, thereby preventing any direct external manipulation.
- There's also reclaim responsibility for RAII class to get ownership back:
  - RAII classes may provide ways to get direct class access to the enclosded resource.
  - RAII classes may even provide ways to break the resource out of the RAII class altogether.

  For example, the `release()` method of `unique_ptr` releases itsownership of the

Takeaway

Since RAII and object lifetime are so intimately intertwined, the following guidelines apply:

- Keep scopes small.
- Always initialize an object.
- Don't introduce a variable (or constant) before you need to use it.
    - Resource management is tied to the lifetime of a variable!
- Don't declare a variable until you have a value to initialize it with.

Smart Pointers

- Smart pointer is wrapper class over a pointer that acts as a pointer but automatically manages the memory it points to. It ensures that memory is properly deallocated when no longer needed, preventing memory leaks. It is a part of header file.
- Smart pointers's cleverness only comes from RAII idiom.
- The C++ libraries provide implementations of smart pointers in the following types:
  - `unique_ptr`, `shared_ptr`, `weak_ptr`

`std::unique_ptr`

- A `unique_ptr` **takes ownership** of a pointer
  - It is a class template: the template parameter `T` is the type that the owned pointer points to (`T*`)
  - Part of C++'s standard library (since C++11)
- Its destructor invokes `delete` on the owned pointer
  - Called when the `unique_ptr` object is destroyed or falls out of scope

Using `unique_ptr`

```cpp
#include <iostream>   // for std::cout, std::endl
#include <memory>     // for std::unique_ptr

void Leaky() {
  int *x = new int(5);    // heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
} // no delete ⇒ leak

void NotLeaky() {
  std::unique_ptr<int> x(new int(5)); // wrapped, heap-allocated
  (*x)++;
  std::cout << *x << std::endl;
} // auto-delete ⇒ no leak

int main() {
  Leaky();
  NotLeaky();
  return 0;
}
```

Why are `unique_ptr`s useful?

- Many exits/returns/exceptions in a function ⇒ easy to forget `delete`
    - `unique_ptr` auto-`delete`s when it falls out of scope
    - Improves **exception safety**

```
void NotLeaky() {
std::unique_ptr<int> x(new int(5));
// …lots of code with early returns or throws…
} // always deletes
```

`unique_ptr` Operations

```cpp
#include <memory>   // std::unique_ptr
#include <cstdlib>  // EXIT_SUCCESS

using namespace std;
typedef struct { int a, b; } IntPair;

int main(int, char**) {
  unique_ptr<int> x(new int(5));

  int *ptr = x.get();   // get raw pointer
  int  val = *x;        // dereference

  unique_ptr<IntPair> ip(new IntPair);
  ip->a = 100;          // member access

  x.reset(new int(1));  // delete old, store new

  ptr = x.release();    // release ownership
  delete ptr;
  return EXIT_SUCCESS;
}
```

Transferring Ownership

- Use `.reset()` and `.release()`
  - `release()` returns the pointer and sets the `unique_ptr` to `nullptr`
  - `reset(p)` deletes the current pointer and takes ownership of `p`

```
#include <memory>
#include <cstdlib>
#include <iostream>
int main(int, char**) {
  unique_ptr<int> x(new int(5));
  cout << "x: " << x.get() << endl;

  unique_ptr<int> y(x.release());     // x abdicates ownership
  cout << "x: " << x.get() << endl;
  cout << "y: " << y.get() << endl;
  unique_ptr<int> z(new int(10));
  z.reset(y.release());               // z takes ownership, deletes old
  return EXIT_SUCCESS;
}
```

`unique_ptr`s Cannot Be Copied

- Copy ctor and copy assignment are **deleted**
  - Enforces unique ownership
- Supports move semantics (C++11)
  - Equivalent to `release()` + `reset()`

```
unique_ptr<int> x(new int(5));
cout << "x: " << x.get() << endl;

unique_ptr<int> y = std::move(x);   // x → y
cout << "x: " << x.get() << endl;
cout << "y: " << y.get() << endl;

unique_ptr<int> z(new int(10));
z = std::move(y);                   // y → z (deletes z's old pointer)
```

`std::shared_ptr`

A `shared_ptr` is similar to `unique_ptr` but allows multiple owners

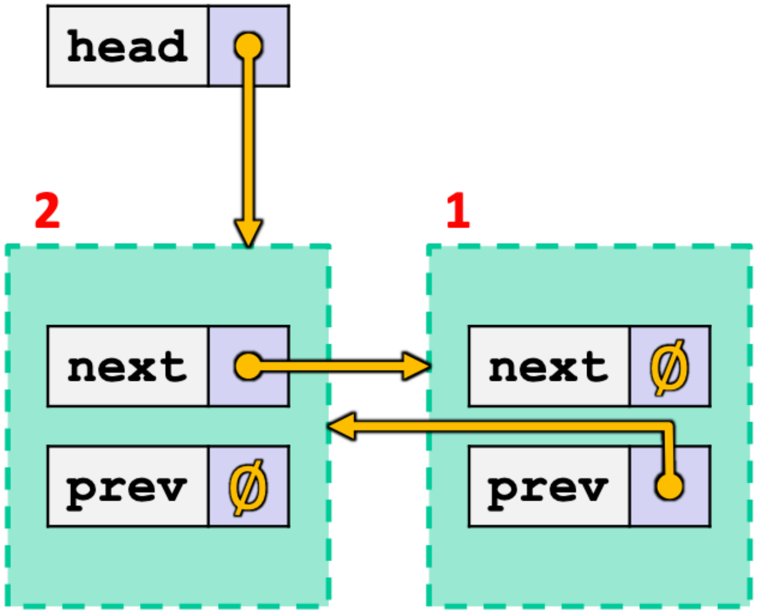
- Copy/assign are **not** deleted → each copy increments the *shared* reference count
    - After copy/assign, both `shared_ptr`s point to the same object; count += 1
- When a `shared_ptr` is destroyed, the reference count is decremented
    - When it reaches 0, the pointed-to object is deleted

`shared_ptr` Example

```cpp
#include <cstdlib>    // EXIT_SUCCESS
#include <iostream>   // std::cout, std::endl
#include <memory>     // std::shared_ptr

int main(int, char**) {
    std::shared_ptr<int> x(new int(10)); // ref count: 1

    { // inner scope
        std::shared_ptr<int> y = x;        // ref count: 2
        std::cout << *y << std::endl;
    } // y is destroyed → ref count: 1

    std::cout << *x << std::endl;        // ref count still: 1
    return EXIT_SUCCESS;                 // x destroyed → ref count: 0 → delete
}
```

Cycle of `shared_ptr`s

```
#include <cstdlib>
#include <memory>

using std::shared_ptr;

struct A {
    shared_ptr<A> next;
    shared_ptr<A> prev;
};

int main(int, char**) {
    shared_ptr<A> head(new A());
    head->next = shared_ptr<A>(new A());
    head->next->prev = head;
    return EXIT_SUCCESS;
}
```



What happens when we delete `head`?

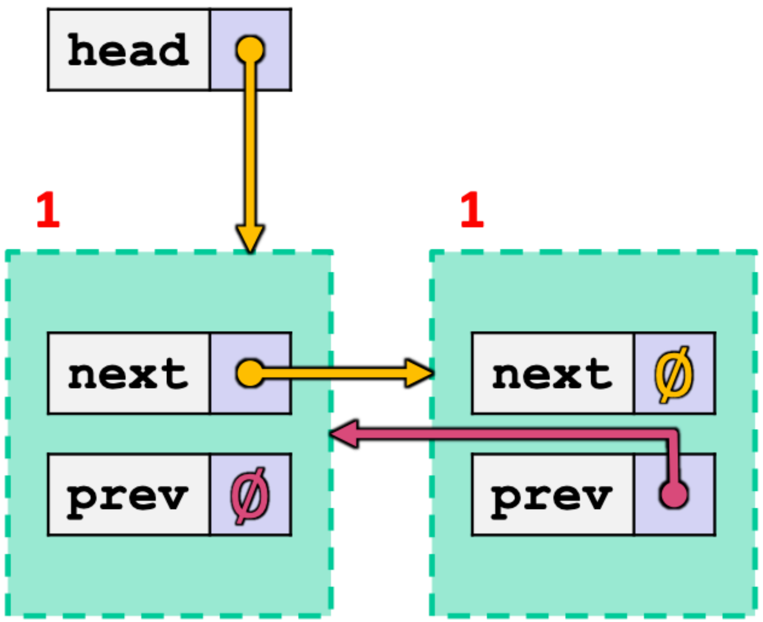
- **Memory leak**: the two `A` objects reference each other, so neither count drops to zero.

`std::weak_ptr`

- A `weak_ptr` is like a non-owning observer of a `shared_ptr`-managed object
  - Does **not** affect reference count
  - Cannot be dereferenced directly
  - Can “point to” only an object managed by `shared_ptr`
  - May dangle if the object is deleted
  - You can check or “promote” it to a `shared_ptr` via `.lock()`

Breaking the Cycle with `weak_ptr`

```
#include <memory>

using std::shared_ptr;
using std::weak_ptr;

struct A {
  shared_ptr<A> next;
  weak_ptr<A> prev;  // break ownership cycle
};

int main(int, char**) {
  shared_ptr<A> head(new A());
  head->next = shared_ptr<A>(new A());

  // prev is weak, does NOT increment count
  head->next->prev = head;

  // both objects can be destroyed properly
  return 0;
}
```



Summary

- `std::unique_ptr`
  - Takes unique ownership of a pointer
  - Cannot be copied, but can be moved (`.release()` , `.reset()` , `std::move`)
- `std::shared_ptr`
  - Shared ownership via reference counting
  - Deletes the object when count reaches zero
- `std::weak_ptr`
  - Non-owning “weak” reference to a `shared_ptr` object
  - Does not affect count; use `.lock()` to get a `shared_ptr` if still alive

CS100 Recitation 10