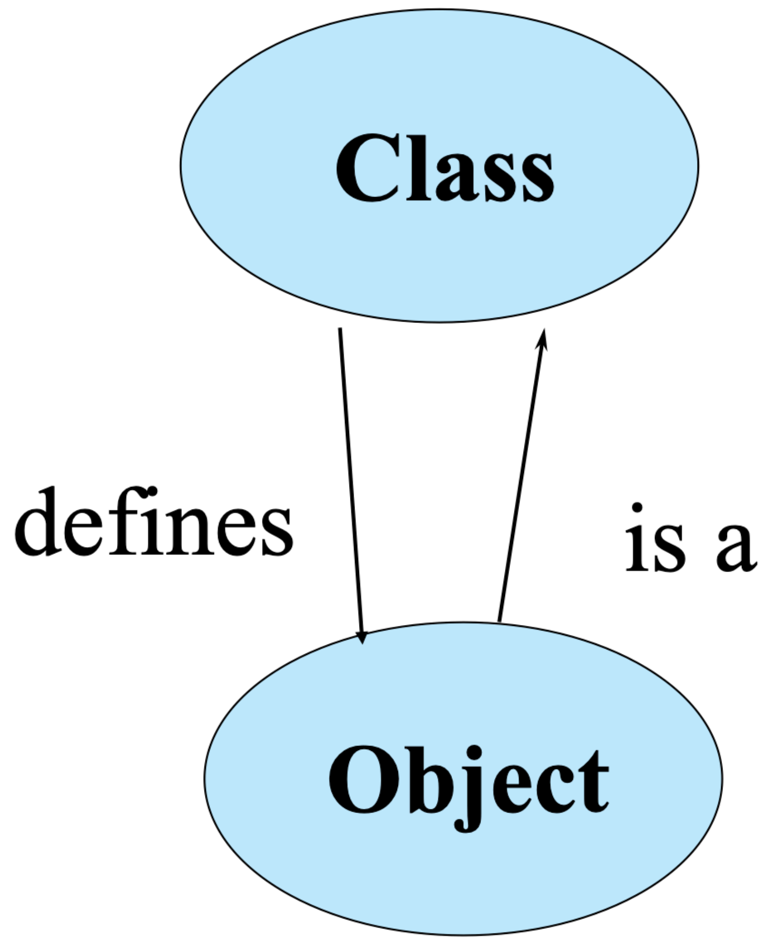
Inside Class

- Overloaded Functions
    - Delegating Constructor
- Default Arguments
- Inline Functions

Object vs. Class

- Object (this cat)
  - Represent things, events, or concepts
  - Respond to messages at run-time
- Classes (the cat)
  - Define properties of instances
  - Act like types in C++



Overloaded constructors

- The constructor has the same name as the class
    - It seems to be the unique one
- Sometimes a default constructor is needed
- But sometimes another constructor with arugments is needed
- Can we have both?
    - Can we have both fish and bear's paw?

Function overloading

- Same functions with different arguments list

```
void print(char * str, int width); // #1
void print(double d, int width); // #2
void print(long l, int width); // #3
void print(int i, int width); // #4
void print(char *str); // #5
print("Pancakes", 15);
print("Syrup");
print(1999.0, 10);
print(1999, 12);
print(1999L, 15);
```

Example: leftover.cpp

Quiz I: Overload and auto-cast

```
void f(short i);
void f(double d);
f('a');
f(2);
f(2L);
f(3.2);
```

Example: overload.cpp

Overloaded `const` and none-const functions

```
void f() const;
void f();
```

Delegating constructors

- If more than one version of the overloaded constructor has to do the same thing inside the constructor
    - Obviously code duplication is a prominent sign of bad design
    - The problem with calling a function inside a constructor is that it happens after initialization.

```
class Info {
public:
    Info() { InitRest();}   //  target
    Info(int i) : Info() { tyep = i; }  // delegating
    Info(char e) : Info() { name=e; }
};
```

- Many classes have multiple constructors that do similar things:

```
class class_c {
public:
    int max;
    int min;
    int middle;
    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Delegating Ctor

- Init members in place can not use aruments for individual object
- Repeated init code in many ctors is a bad design -- code duplication
- Put calling to another ctor in the initialization list.
- But this delegating ctor can not have other members initialized in its own initialization list
- To solve this, a private ctor can be used to provide initialization to other members.
- It is possible to create a chain of delegating ctors.

Delegating Ctor

```cpp
class class_c {
public:
    int max;
    int min;
    int middle;
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
}
};
int main() {
    class_c c1{ 1, 3, 2 };
}
```

- The execution of the target constructor precedes that of the delegate constructor.
- Constructors cannot delegate and use initialization lists at the same time.
    - Assigning values inside a constructor is not the best choice
    -

```
class Info {
public:
    Info(): Info(1, 'a') {}   //  delegating
    Info(int i) : Info(i, 'a') {}  // delegating
    Info(char e) : Info(1, e) {  }
private:
    Info(int i, char e): type(i), name(e) {}    //  target
};
```

- Delegation relationships can form links
-

```
class Info {
public:
    Info(): Info(1) {}   //  delegating
    Info(int i) : Info(i, 'a') {}  // target & delegating
    Info(char e) : Info(1, e) {  }
private:
    Info(int i, char e): type(i), name(e) {}    //  target
};
```

- But circular links should get prevented.

Default arguments

```
Stash(int size, int initQuantity = 0);
```

- A default argument is a value given in the declaration that the compiler automatically inserts if you don’t provide a value in the function call.

```
int harpo(int n, int m = 4, int j = 5);
int chico(int n, int m = 6, int j);//illeagle
int groucho(int k = 1, int m = 2, int n = 3);
beeps = harpo(2);
beeps = harpo(1,8);
beeps = harpo(8,7,6);
```

Example: left.cpp

- To define a function with an argument list, defaults must be added from right to left.

Ctor with default argument

- A ctor with all arguments default is a default ctor

Pitfall of default arguments

- The default arguments are declared in its prototype, not defined in its function head
    - Can not put default arguments in definition
    - May be changed by ill-formed prototype

Inline Functions

Overhead for a function call

- the processing time required by a device prior to the execution of a command
    - Push parameters
    - Push return address

      …
    - Prepare return values
    - Pop all pushed

Inline function

- An inline function is to be expanded in place, like a preprocessor macro, so the overhead of the function call is eliminated.

```
int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

```
inline int f(int i) {
    return i*2;
}
main() {
    int a=4;
    int b = f(a);
}
```

Inline Functions

```
inline int plusOne(int x);
inline int plusOne(int x) {return ++x; };
```

- Repeat inline keyword at declaration and definition.
- An inline function definition may not generate any code in .obj file.

Inline functions in header file

- So you **must** put inline functions' bodies in header file. Then `#include` it where the function is needed.
- Never be afraid of multi-definition of inline functions. Definitions of inline functions are just declarations.
  - since they have no body at all.

Tradeoff of inline functions

- Body of the called function is to be inserted into the caller.
- This may expand the code size but deduces the overhead of calling time. So it gains speed at the expenses of space. In most cases, it is worth.
- It is much better than macro in C. It checks the types of the parameters.

```
#define f(a) (a)+(a)
main() {
    double a=4;
    printf("%d",f(a));
}
```

```
inline int f(inti) {
    return i*2;
}
main() {
    double a=4;
    printf("%d",f(a));
}
```

Example: inline1.cpp

Inline may not in-line

- The compiler does not have to honor your request to make a function inline. It might decide the function is too large or notice that it calls itself (recursion is not allowed or indeed possible for inline functions), or the feature might not be implemented for your particular compiler.

Inline inside classes

- Any function you define inside a class declaration is automatically an inline.

Example: Inline.cpp

Pit-fall of inline

- You can put the definition of an inline member function out of the class braces.
- But the definition of the functions should be put before where they may be called, in this case, in the header file.

Example: NotInline.h, NotInline.cpp, NotInlineTest.cpp

Reducing clutter

- Member functions defined within classes use the in situ (in place) and maintains that all definitions should be placed outside the class to keep the interface clean.

Example: Noinsitu.cpp

Inline or not?

- Inline:
  - Small functions, 2 or 3 lines
  - Frequently called functions, e.g. inside loops
- Not inline?
  - Very large functions, more than 20 lines
  - Recursive functions
- A lazy way
  - Make all your functions inline, or
  - Never make your functions inline

`inline` Variables in C++

In C++17 and later, the `inline` keyword can be applied to **variables** in addition to functions. `inline` variables solve a common issue related to **global variables** and **constant definitions** in header files.

Why `inline` Variables Were Introduced

Before C++17, defining a variable in a header file caused issues because each translation unit (i.e., `.cpp` file) that included the header would generate a separate copy of the variable. This often led to **multiple definition errors** during the linking phase.

To avoid this, variables were typically declared using `extern` in header files and defined in a single `.cpp` file:

```
// header.h
extern int globalValue;

// source.cpp
int globalValue = 42;
```

This worked but required splitting the declaration and definition, which wasn't always convenient.

`inline` Variable Solution

With `inline` variables, you can define variables directly in header files without causing linker errors.

Key Properties of `inline` Variables

1. **Single Definition Across Translation Units**:
   The compiler ensures only **one instance** of an `inline` variable exists, even if the variable is included in multiple files.
2. **Initialization in Header Files**:
   You can both declare and initialize the variable in a header, making code more concise.
3. **Global or Static Variables**:
   Works well with global constants, static members of classes, or singleton patterns.

Example 1: Using `inline` Variables

```
// constants.h
#ifndef CONSTANTS_H
#define CONSTANTS_H

inline int globalValue = 42;
inline const double pi = 3.14159;

#endif
```

```
// main.cpp
#include <iostream>
#include "constants.h"

int main() {
    std::cout << "Global Value: " << globalValue << "\n";
    std::cout << "Pi: " << pi << "\n";
    return 0;
}
```

Example 2: `inline` with Class Static Members

```cpp
class Config {
public:
    inline static int maxConnections = 100;
    inline static std::string appName = "MyApp";
};
int main() {
    std::cout << "Max Connections: " << Config::maxConnections << "\n";
    std::cout << "App Name: " << Config::appName << "\n";
    return 0;
}
```

Explanation:

- `inline static` allows defining and initializing static class members directly in the class definition.
- No need for a separate definition in a `.cpp` file.

When to Use `inline` Variables

- When defining **global constants** in header files.
- When using **static class members** that are initialized directly in the class definition.
- When needing a **single definition** of a variable across multiple files.
- When simplifying the management of configuration values or settings.

When Not to Use `inline` Variables

- Avoid using `inline` variables for **large objects** or variables that consume significant memory. It may lead to unnecessary memory usage.
- Don’t use `inline` for variables that **require unique instances** in each translation unit.
- Be cautious with `inline` for variables that can be **modified** at runtime, as this can lead to unintended side effects.

Conclusion

- `inline` variables simplify the management of global variables and static members in C++17 and later.
- They provide a cleaner, safer alternative to `extern` for constants and settings.
- Just remember, `inline` doesn't mean the variable is copied — it ensures a **single definition** across all translation units.

Weak

`inline` and `weak` are two different keywords, used for distinct purposes, mainly involving functions and variables.

- Linkage Involvement:
  Both keywords are related to symbol management during the linking phase of the compilation process.
- Multiple Definitions Allowed:
  They both allow multiple definitions under certain conditions without causing linker errors.

Aspect	`inline`	`weak`
Primary Purpose	Optimizes function calls by suggesting inlining to the compiler.	Allows symbol overriding and prevents linker conflicts.
Applicable To	Primarily used for functions, especially small or frequently called ones.	Used for functions, variables, or objects, especially in dynamic linking or plugin systems.
Symbol Management	The compiler may generate multiple copies of the function across translation units and deduplicate them.	Multiple weak symbols can coexist, and the linker chooses the strongest (non-weak) one if available.

Aspect	`inline`	`weak`
Linker Behavior	No linker error occurs if the same `inline` function is defined in multiple translation units.	No linker error occurs if multiple weak symbols exist; a non-weak symbol is preferred if present.
Optimization Focus	Focuses on reducing function call overhead for performance improvement.	Provides fallback implementations or optional symbols for plugins or dynamic libraries.
Use Case	Best suited for small, simple functions, template functions, or class member functions.	Commonly used for default implementations, modular plugins, or singleton patterns.

Example: Using `weak`

```
// Weak function definition as a fallback
__attribute__((weak)) void printMessage() {
    std::cout << "Default message" << std::endl;
}
```

Explanation:

- If no other implementation of `printMessage()` exists, this weak version will be used.
- If another non-weak definition is provided in a separate translation unit, the non-weak version will take precedence.
- Useful for providing optional or fallback functionality in modular applications.

Summary

- Use `inline` to **improve performance** by avoiding function call overhead for small functions.
- Use `weak` to **handle symbol conflicts** or provide optional implementations, particularly in dynamic or plugin-based environments.
- While both involve symbol management, `inline` focuses on performance, whereas `weak` focuses on flexibility and compatibility during linking.

What we've learned?

- Overloaded functions
- Default arguments
- Inline functions