

# Object-Oriented Programming

Reference Types

# Recap – Object Oriented Programming

- Model software through **objects and their interaction**
- **Object** = self-contained units that combines attributes (data) and methods (behavior)
- Object **attributes / elements** = data
- Object **methods** = functions that operate on object's internal data
- Access to methods and attributes can be public or private to objects

# Recap – Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

# Recap – Encapsulation

- Access to private members and methods is only possible over public interface
  - With access specifiers **public** / **private**
- Prevents read and write access from outside the class
- Member functions can access all members of their own class

```
1  class Pet {  
2      private: // from here on, everything is private  
3          unsigned age_;  
4  
5      public: // from here on, everything is public  
6          void setAge(unsigned age) {  
7              age_ = age;  
8          }  
9  };
```

is the age someone is setting  
valid?

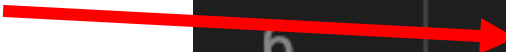
# Recap – Encapsulation

- Hide internal details and protects the integrity of an object's state.
- Encapsulation provides **controlled access** and **protects data**
- Better Reuse in other projects
  - easier usage of the class
- Improved Maintainability
  - changing the private part of a class has no side effects outside of the class

# Setter

- We call setAge a **setter**

```
1  class Pet {  
2      private: // from here on, everything is private  
3          unsigned age_;  
4  
5      public: // from here on, everything is public  
6          void setAge(unsigned age) {  
7              age_ = age;  
8          }  
9  };
```



# Setter


- We call setAge a **setter**
- For writing to private attributes
- Usually 1 parameter, no return value, Name: setAttributeName

```
1  class Pet {
2      private: // from here on, everything is private
3          unsigned age_;
4
5      public: // from here on, everything is public
6          void setAge(unsigned age) {
7              age_ = age;
8          }
9  };
```

# Setter

- call `setAge(..)` on object to set attribute value

```
1  class Pet {  
2      private: // from here on, everything is private  
3          unsigned age_;  
4  
5      public: // from here on, everything is public  
6          void setAge(unsigned age) {  
7              age_ = age;  
8          }  
9  };  
10  
11  int main() {  
12      Pet pet{};  
13      pet.setAge(0); // OK  
14      return 0;  
15  }
```






# The Constructor

- Special method - automatically called when the object is created
- Name of constructor = name of class
- no return value


```
1  class Pet {
2      private: // from here on, everything is private
3          unsigned age_;
4
5      public: // from here on, everything is public
6          void setAge(unsigned age) {
7              age_ = age;
8          }
9  };
10
11 int main() {
12     Pet pet{};
13     pet.setAge(0); // OK
14     return 0;
15 }
```



# The Default-Constructor

- Compiler adds **Default-Constructor**, if you don't write one.
- The Default-Constructor has no parameters and initializes nothing.

```
1  class Pet {
2      private: // from here on, everything is private
3          unsigned age_;
4
5      public: // from here on, everything is public
6          void setAge(unsigned age) {
7              age_ = age;
8          }
9  };
10
11  int main() {
12      Pet pet{};
13      pet.setAge(0); // OK
14      return 0;
15  }
```



# The Constructor

- Can have multiple constructors
- Must have different parameters

```
1  class Pet {  
2      private:  
3          unsigned age_;  
4      public:  
5          Pet() { age_ = 0; }  
6          Pet(unsigned age) { age_ = age; }  
7          unsigned getAge() {  
8              return age_;  
9          }  
10 };  
  
11  
12 int main() {  
13     Pet pet{2}; // constructor sets pet.age_ to 2.  
14     return 0;  
15 }
```

overloading  
→ multiple fitting calls

parameters

# The Default-Constructor

- Will not be generated if another constructor exists !

```
1  class Pet {
2      private:
3          unsigned age_;
4      public:
5          Pet() { age_ = 0; }
6          Pet(unsigned age) { age_ = age; }
7          unsigned getAge() {
8              return age_;
9          }
10 };
11
12 int main() {
13     Pet pet{2}; // constructor sets pet.age_ to 2.
14     return 0;
```

# The Default-Constructor

- Will not be generated if another constructor exists !

```
1  class Pet {
2  private:
3      unsigned age_;
4  public:
5      Pet(unsigned age) : age_{age} {}
6  };
7
8  int main() {
9      Pet pet{}; // error: no matching function for call to 'Pet::Pet()'
10     return 0;
11 }
```


*default constructor missing*



# The Default-Constructor

- We can add a Default-Constructor manually

```
1  class Pet {  
2  private:  
3      unsigned age_;  
4  public:  
5      Pet() = default; // similar to: Pet() {}  
6      Pet(unsigned age) : age_{age} {}  
7  };  
8  
9  int main() {  
10     Pet pet{}; // OK  
11     return 0;  
12 }
```



# The Default-Constructor

- **Problem:** age\_ is not initialized by default constructor

```
1  class Pet {  
2  private:  
3      unsigned age_;  
4  public:  
5      Pet() = default; // similar to: Pet() {}  
6      Pet(unsigned age) : age_{age} {}  
7  };  
8  
9  int main() {  
10     Pet pet{}; // OK  
11     return 0;  
12 }
```

# The Default-Constructor

- Possible solution: Custom constructor without parameters but with initial values of attributes

```
1  class Pet {  
2  private:  
3      unsigned age_;  
4  public:                                initialize list  
5      Pet() : age_{0} {} ←  
6      Pet(unsigned age) : age_{age} {}  
7  };  
8  
9  int main() {  
10     Pet pet{}; // OK ←  
11     return 0;  
12 }
```



# The Default-Constructor

- Other possibility: Default values for attribute



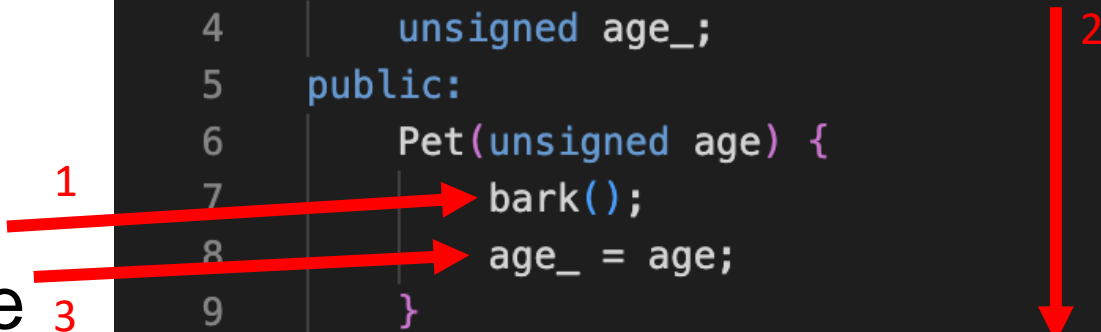
```
1  class Pet {
2  private:
3      unsigned age_ = 0; // also possible: unsigned age_{0}
4  public:
5      Pet() = default; // similar to: Pet() {}
6      Pet(unsigned age) : age_{age} {}
7  };
8
9  int main() {
10     Pet pet{}; // OK
11     return 0;
12 }
```

# The Constructor

## Problem 1

- age\_ is used before initialization (default value is random)

```
1  #include <iostream>
2
3  class Pet {
4      unsigned age_;
5  public:
6      Pet(unsigned age) {
7          bark();
8          age_ = age;
9      }
10     void bark() { std::cout << age_; }
11 };
12
13 int main() {
14     Pet pet{2}; // constructor sets pet.age_ to 2.
15     return 0;
16 }
```




# The Constructor

## Problem 2

- Can be inefficient:
  1. Create empty string
  2. copy content to override empty string

**redundant !**


```
1  #include <iostream>
2
3  class Pet {
4      std::string name_;
5      Pet(std::string name) {
6          name_ = name;
7      }
8  };|
```



# The Constructor – Initializer List

- **‘Initializer list’** avoids both problems


```
1  class Pet {  
2      unsigned age_;  
3  
4  public:  
5      Pet() : age_{0} {}  
6      Pet(unsigned age) : age_{age} {}  
7  };  
8  
9  int main() {  
10     Pet pet{}; // OK  
11     return 0;  
12 }
```



# The Constructor – Initializer List

- With multiple attributes:

```
Pet(char* name, unsigned age)
    : name_{name}, age_{age}
    {}
```



# Summary

- Initialize objects with **Constructor**
- Use '**Setter**' for setting attribute values

# Setter


- Setters can do more
- e.g. logging or validating

```
1  #include <stdio>
2
3  class Pet {
4      private:
5          unsigned age_;
6      public:
7          void setAge(unsigned age) {
8              if (age > 1000)
9                  printf("A stone is not a pet.");
10             else
11                 age_ = age;
12         }
13 };
```

default constructor will be created if not defined

```
1  #include <stdio>
2
3  class Pet {
4      private:
5          unsigned age_;
6      public:
7          void setAge(unsigned age) {
8              if (age > 1000)
9                  printf("A stone is not a pet.");
10             else
11                 age_ = age;
12         }
13     };
```

What is the output?



```
int main() {
    Pet pet{};
    pet.setAge(0);
    pet.setAge(2000);
    printf("Your pet's age: %u", pet.age_); // POLL
    return 0;
}
```

Compiler error  
as age\_ is private.




# Getter

- How to access attributes from outside? => **Getter**
- Usually 0 parameters, attribute as return value, Name: getAttrName


```
1  #include <stdio>
2
3  class Pet {
4      private:
5          unsigned age_;
6      public:
7          void setAge(unsigned age) {
8              if (age > 1000)
9                  printf("A stone is not a pet.");
10             else
11                 age_ = age;
12         }
13         unsigned getAge() {
14             return age_;
15         }
16     };

```



What is the output?

```
1  #include <stdio>
2
3  class Pet {
4      private:
5          unsigned age_;
6      public:
7          void setAge(unsigned age) {
8              if (age > 1000)
9                  printf("A stone is not a pet.");
10             else
11                 age_ = age;
12         }
13         unsigned getAge() {
14             return age_;
15         }
16     };
17
18     int main() {
19         Pet pet{};
20         pet.setAge(0);
21         pet.setAge(2000);
22         printf("Your pet's age: %u", pet.getAge()); // POLL
23         return 0;
24     }
```



# Deleting the Default-Constructor

- Default-Constructor can be deleted explicitly
- Object cannot be created, if there is no constructor

```
1  class Pet {
2  private:
3      unsigned age_;
4  public:
5      Pet() = delete;
6  };
7
8  int main() {
9      Pet pet{}; // error: no matching function for call to 'Pet::Pet()'
10     return 0;
11 }
12
```

# Initialization

- **MyClass obj;** // Default initialization
- ... calls the **default constructor** of MyClass (if it exists).

# Initialization

- **MyClass obj(42);** // parenthesis initialization (direct initialization)
- Calls a constructor whose parameters match the argument types.

# Initialization

- **MyClass obj(42);** // parenthesis initialization (direct initialization)
- Calls a constructor whose parameters match the argument types.
- ... calls Constructor **MyClass(int value){...}**

# Initialization

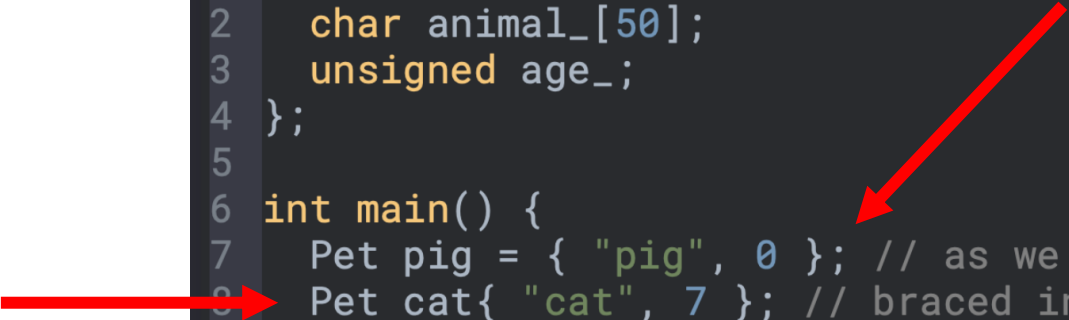
- Braced initializer

```
1  class Pet {  
2      unsigned age_;  
3  
4  public:  
5      Pet() : age_{0} {}  
6      Pet(unsigned age) : age_{age} {}  
7  };  
8  
9  int main() {  
10     Pet pet{}; // OK  
11     return 0;  
12 }
```

# Recap C - Initialization of PODs

- Braced initializer

```
1 struct Pet {  
2     char animal_[50];  
3     unsigned age_;  
4 };  
5  
6 int main() {  
7     Pet pig = { "pig", 0 }; // as we know it from C  
8     Pet cat{ "cat", 7 }; // braced initializer (=uniform initializer)  
9 }
```





# Uniform initialization

- since C++11
- **Braced initializer**
- Example: `MyClass obj2{42};`

# Uniform initialization

- Besides PODs: Uniform initialization can also be used for primitive initialization
- For example: `int x{5};`      `// primitive`

# Uniform initialization

- Uniform initialization avoids *narrowing*!

```
int a = 3.14;    // truncates to 3, no warning!
```

- Better => Error protects from narrowing from double to int
- {} gives **compile-time safety** against unintended data loss


```
int a{3.14};    //
```

# Problem Direct Initialization – Ambiguity

- Uniform initialization avoids ambiguity
- Default constructor vs. function declaration

```
class InitTester {  
public:  
    InitTester()        { printf("default - no argument\n"); }  
    InitTester(char c)  { printf("char %c\n", c); }  
    InitTester(int i)   { printf("int %d\n", i); }  
    InitTester(float f) { printf("float %f\n", f); }  
};
```

```
int main() {  
    InitTester t1; → default  
    InitTester t2{ 'c' }; → char  
    InitTester t3{ 65537 }; → int  
    InitTester t4{ 60.1f }; → float  
    InitTester t5('g'); → char  
    InitTester t6 = { 'l' }; → char  
    InitTester t7{}; → default.  
    InitTester t8(); → function declaration  
}
```



# Pointer and References

# Memory Address

- Variables and functions are stored in memory
- Have an **address** in memory
- Address can be accessed using the address-of-operator **&**
- Example: `int number=1;`  
    `&number => Address of number`

```
1  #include <stdio>
2
3  int main() {
4      int number = 1;
5      int *ptr = &number;
6      printf("printf.1: %p\n", &number); //0x7fffffffcdcd
7      printf("printf.2: %p\n", ptr); //0x7fffffffcdcd
8      return 0;
9  }
```

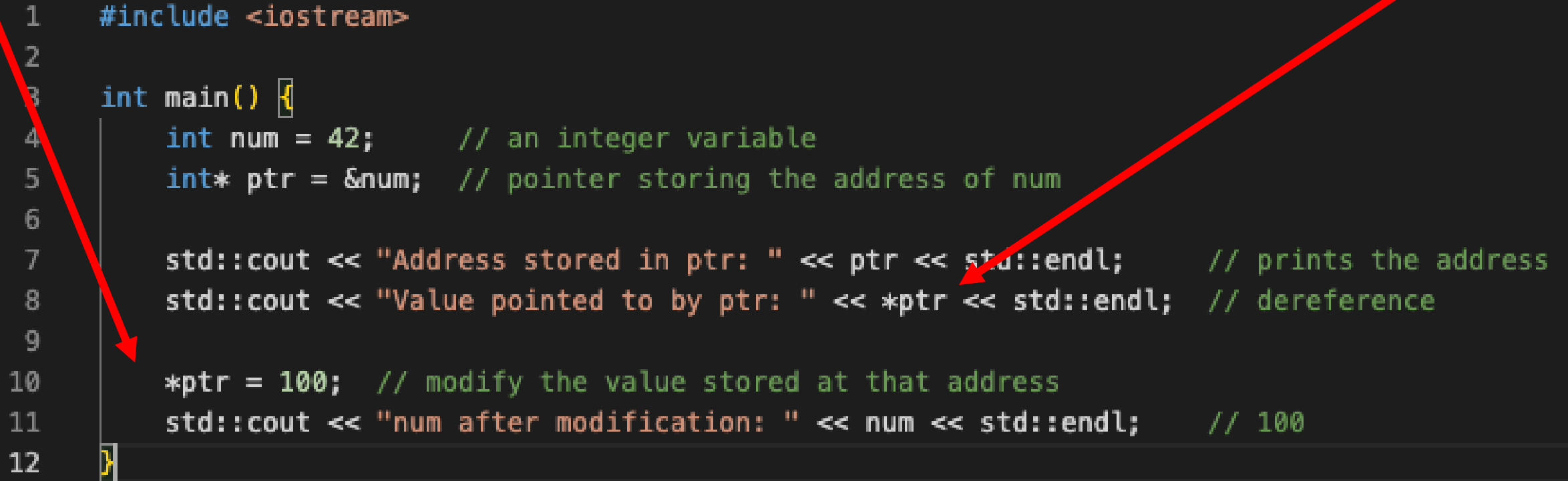
# Pointer

```
1  #include <stdio>
2
3  int main() {
4      int number = 1;
5      int *ptr = &number;
6      printf("printf.1: %p\n", &number); //0x7fffffffcdcd
7      printf("printf.2: %p\n", ptr); //0x7fffffffcdcd
8      return 0;
9  }
```

- **A pointer** is a variable that **stores a memory address**
- Example: **int\* a\_ptr = &a;**

# De-Reference Pointer

- Use `*ptr` to access the value pointed to



```
1  #include <iostream>
2
3  int main() {
4      int num = 42;    // an integer variable
5      int* ptr = &num; // pointer storing the address of num
6
7      std::cout << "Address stored in ptr: " << ptr << std::endl;    // prints the address
8      std::cout << "Value pointed to by ptr: " << *ptr << std::endl; // dereference
9
10     *ptr = 100; // modify the value stored at that address
11     std::cout << "num after modification: " << num << std::endl;    // 100
12 }
```

The code demonstrates pointer dereferencing. A red arrow points from the top left to the opening curly brace of the `main` function. Another red arrow points from the top right to the `*ptr` expression in the second `std::cout` statement, illustrating the dereferencing operation.



# Special Pointer

```
1 class Cat
2 {
3     unsigned age;
4     public:
5         Cat(unsigned age) {age = age;}
6 };
```



Compiler output

```
warning: explicitly assigning value of variable of type 'unsigned int' to itself
Cat(unsigned age) {age = age;}
      ~~~ ^ ~~~
```

Better: use naming convention

```
1 class Cat
2 {
3     unsigned age_;
4     public:
5         Cat(unsigned age) { age_ = age; }
6 };
```

Two red arrows originate from the top right of the image. One arrow points diagonally down and to the left towards the variable name 'age\_' on line 3. The other arrow points diagonally down and to the left towards the parameter 'unsigned age' in the constructor on line 5.

## Solution 2: Initialization list

```
1 class Cat
2 {
3     unsigned age;
4     public:
5         Cat(unsigned age) : age{age} {}
6 };
```



# Special Pointer

Solution 3: use special pointer **this**

- points to the object on which the method was called
- use -> to call methods or to access an object's variable
- available in every non-static method (explanation of static follows)


```
1 class Cat
2 {
3     unsigned age;
4     public:
5         Cat(unsigned age) { this->age = age; }
6 };
```



# Method chaining with this-Pointer

```
1 class Student {  
2     public:  
3     Student& code() { printf("coding is fun\n"); return *this; }  
4     Student& sleep() { printf("zzz\n"); return *this; }  
5 };  
6  
7 int main() {  
8     Student linus{};  
9     linus.code().sleep().code().sleep(); // fun weekend  
10 }
```

*reference to student; similar to pointer*



# Method chaining with this-Pointer

```
1 class Student {
2     public:
3         Student& code() { printf("coding is fun\n"); return *this; }
4         Student& sleep() { printf("zzz\n"); return *this; }
5 };
6
7 int main() {
8     Student linus{};
9     linus.code().sleep().code().sleep(); // fun weekend
10 }
```



Output:

```
coding is fun
zzz
coding is fun
zzz
```

# Method chaining with this-Pointer

- Possible by returning `*this`
- Advantage: only one statement (instead of four in this example)

```
1 class Student {  
2     public:  
3         Student& code() { printf("coding is fun\n"); return *this; }  
4         Student& sleep() { printf("zzz\n"); return *this; }  
5 };  
6  
7 int main() {  
8     Student linus{};  
9     linus.code().sleep().code().sleep(); // fun weekend  
10 }
```





# Special Pointer - NULL

- In C macro **NULL** points to 0L
- Represented as **int**

# Special Pointer - NULL

- Represented as **int** => Can be ambiguous

```
1 void function(int arg) {  
2     printf("arg is an integer\n");  
3 }  
4 void function(int* arg) {  
5     printf("arg is a pointer\n");  
6 }  
7  
8 int main() {  
9     function(NULL); // what is the output?  
10    return 0;  
11 }
```

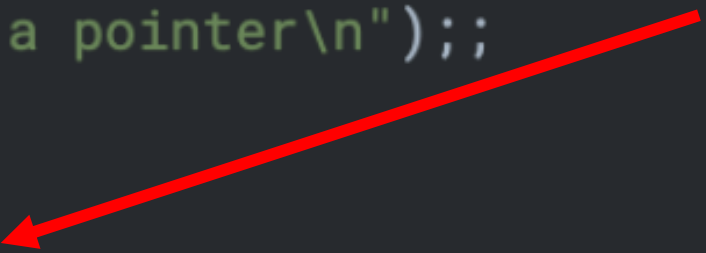
→ long 32 bit

→ long 64 bit

# Special Pointer - nullptr

- Better use nullptr
- Is of type std::nullptr\_t
- Can be implicitly converted to any pointer data type

```
1 void function(int arg) {  
2     printf("arg is an integer\n");  
3 }  
4 void function(int* arg) {  
5     printf("arg is a pointer\n");  
6 }  
7  
8 int main() {  
9     function(nullptr); // > arg is a pointer  
10    return 0;  
11 }
```



# Special Pointer - nullptr

- Every pointer can be implicitly converted to bool
- The result is false if the pointer is **nullptr** and true otherwise.

```
if (nullptr)
    printf("never ever\n");
else
    printf("nullptr is false!\n");
```

# Special Pointer - nullptr

- **Cannot** implicitly converted into another data type

```
int number = nullptr; // error: cannot convert std::nullptr_t to int
```

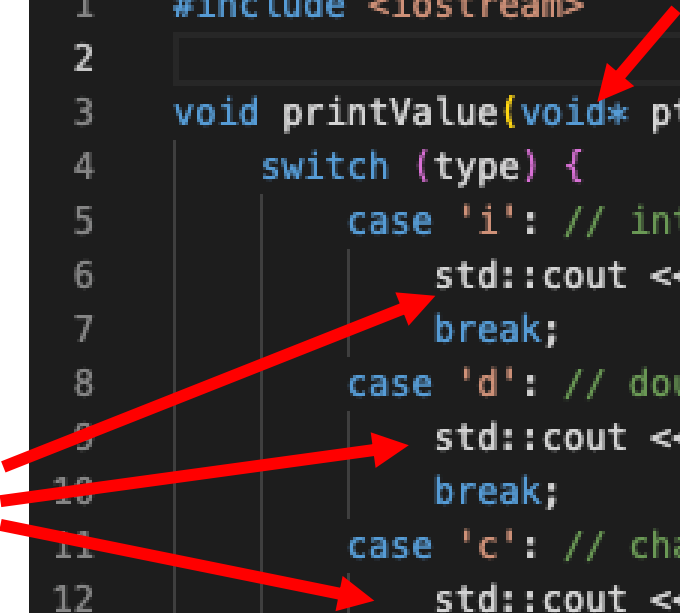
# Special Pointer – void\*

- generic pointer type
- can hold the address of any data type

# Example – void\*

- can hold the address of any data type
- void\* can be cast to specific type

```
1  #include <iostream>
2
3  void printValue(void* ptr, char type) {
4      switch (type) {
5          case 'i': // int
6              std::cout << *(int*)ptr << std::endl;
7              break;
8          case 'd': // double
9              std::cout << *(double*)ptr << std::endl;
10             break;
11          case 'c': // char
12              std::cout << *(char*)ptr << std::endl;
13              break;
14          default:
15              std::cout << "Unknown type!\n";
16      }
17  }
18
19  int main() {
20      int a = 42;
21      double b = 3.14;
22      char c = 'Z';
23
24      printValue(&a, 'i');
25      printValue(&b, 'd');
26      printValue(&c, 'c');
27  }
```




The diagram illustrates the casting of a void\* pointer to specific data types within the printValue function. Red arrows point from the void\* parameter to the casted pointers in the switch cases: (int\*)ptr for 'i', (double\*)ptr for 'd', and (char\*)ptr for 'c'. This demonstrates how void\* can be cast to any data type.

# Warning!

- If you pass the wrong type code, you'll get **undefined behavior**
- For example:

```
printValue(&a, 'd');  
// treated as double  
// undefined behavior!
```

```
1  #include <iostream>  
2  
3  void printValue(void* ptr, char type) {  
4      switch (type) {  
5          case 'i': // int  
6              std::cout << *(int*)ptr << std::endl;  
7              break;  
8          case 'd': // double  
9              std::cout << *(double*)ptr << std::endl;  
10             break;  
11          case 'c': // char  
12              std::cout << *(char*)ptr << std::endl;  
13              break;  
14          default:  
15              std::cout << "Unknown type!\n";  
16          }  
17 }  
18  
19 int main() {  
20     int a = 42;  
21     double b = 3.14;  
22     char c = 'Z';  
23  
24     printValue(&a, 'i');  
25     printValue(&b, 'd');  
26     printValue(&c, 'c');  
27 }
```





# Warning!


- If you pass the wrong type code, you'll get **undefined behavior**

- For example:

```
printValue(&a, 'd');  
// treated as double  
// undefined behavior!
```

- Avoid if possible - because the compiler can't catch mistakes

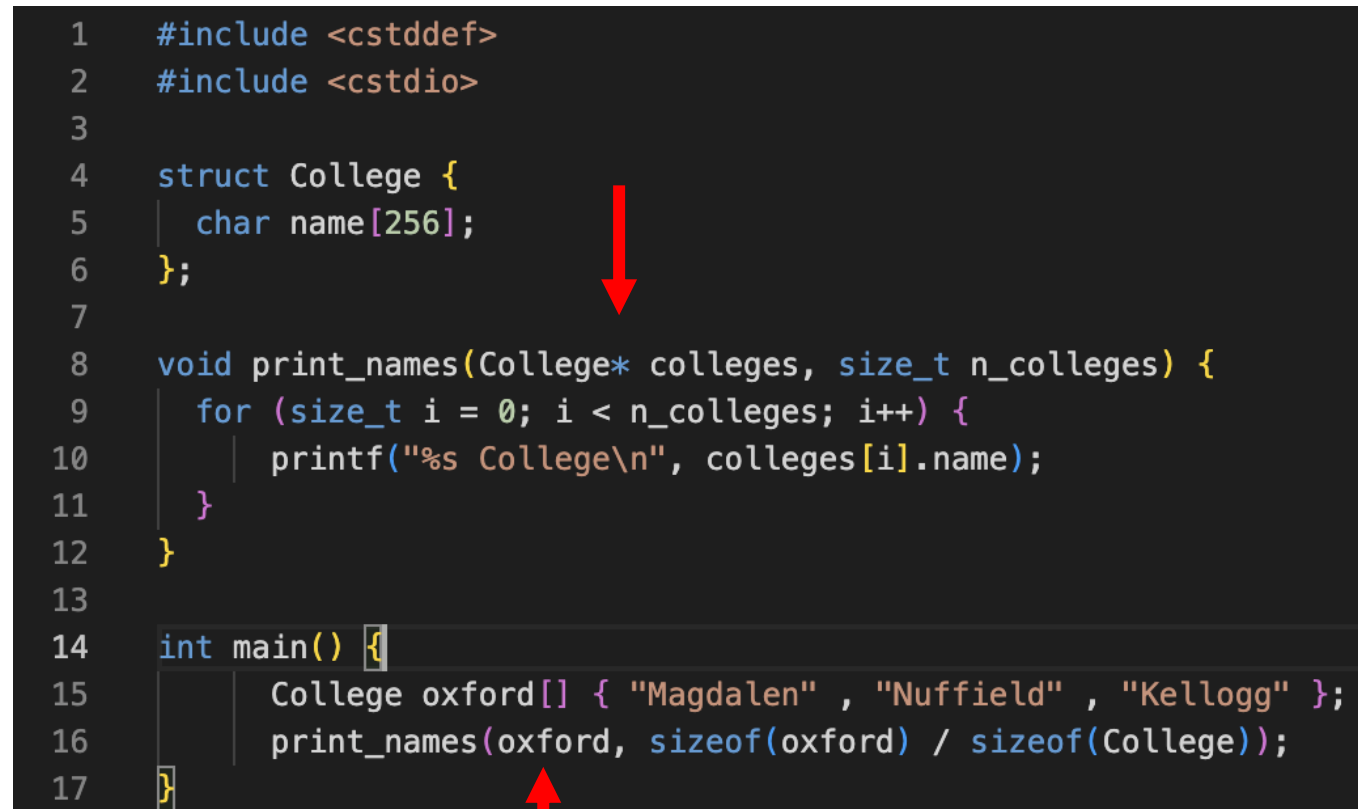
```
1  #include <iostream>  
2  
3  void printValue(void* ptr, char type) {  
4      switch (type) {  
5          case 'i': // int  
6              std::cout << *(int*)ptr << std::endl;  
7              break;  
8          case 'd': // double  
9              std::cout << *(double*)ptr << std::endl;  
10             break;  
11          case 'c': // char  
12              std::cout << *(char*)ptr << std::endl;  
13              break;  
14          default:  
15              std::cout << "Unknown type!\n";  
16          }  
17 }  
18  
19 int main() {  
20     int a = 42;  
21     double b = 3.14;  
22     char c = 'Z';  
23  
24     printValue(&a, 'i');  
25     printValue(&b, 'd');  
26     printValue(&c, 'c');  
27 }
```



# Pointer for Array Decay

- When we use an array name, it is automatically converted to (or “decays into”) a pointer
- `College* colleges = &oxford[0];`

```
1  #include <csddef>
2  #include <stdio>
3
4  struct College {
5      char name[256];
6  };
7
8  void print_names(College* colleges, size_t n_colleges) {
9      for (size_t i = 0; i < n_colleges; i++) {
10         printf("%s College\n", colleges[i].name);
11     }
12 }
13
14 int main() {
15     College oxford[] { "Magdalen" , "Nuffield" , "Kellogg" };
16     print_names(oxford, sizeof(oxford) / sizeof(College));
17 }
```


A diagram illustrating array decay. A red arrow points from the array name 'oxford' in the function call on line 16 to the 'College\*' parameter in the function signature on line 8. Another red arrow points from the 'College\*' parameter on line 8 to the 'char name[256];' declaration on line 5, indicating that the array name decays into a pointer to the first element of the array.

# Pointer arithmetic

- ++ operator can also **increment a pointer**  
=> increment points to the **next element of its type**

- college++  
=> college = college  
+ 1 \* sizeof(College)

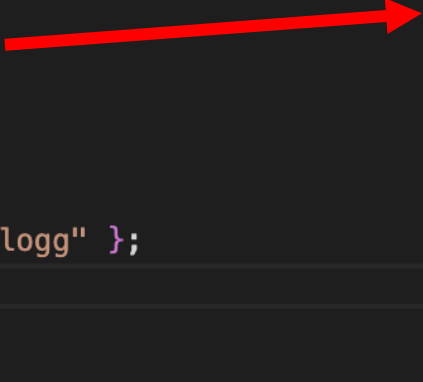
```
1  #include <csddef>
2  #include <csdio>
3
4  struct College {
5      char name[256];
6  };
7
8  void print_names(College* colleges, size_t n_colleges) {
9      for (College* college = colleges; college < colleges + n_colleges; college++) {
10         printf("%s College\n", college->name);
11     }
12 }
13
14 int main() {
15     College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
16     print_names(oxford,3);
17
18     return 0;
19 }
```



# Pointer arithmetic

- Can be dangerous
- Random result or Prog. Crash

```
1  #include <csddef>
2  #include <csdio>
3
4  struct College {
5      char name[256];
6  };
7
8  void print_names(College* colleges, size_t n_colleges) {
9      for (College* college = colleges; college < colleges + n_colleges; college++) {
10         printf("%s College\n", college->name);
11     }
12 }
13
14 int main() {
15     College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
16     print_names(oxford,4);
17
18     return 0;
19 }
```




# Call-by-Value

- A copy is passed

```
1  #include <iostream>
2
3  void addTen(int x) {
4      x = x + 10;  // only changes the copy
5  }
6
7  int main() {
8      int num = 5;
9      addTen(num);    // pass by value
10     std::cout << num << std::endl; // prints 5, not 15
11 }
```

# Call-by-Reference


- When called **with a reference**, the actual variable is passed



```
1  #include <iostream>
2
3  void addTen(int &x) {
4      x = x + 10; // changes the original variable
5  }
6
7  int main() {
8      int num = 5;
9      addTen(num); // pass by reference
10     std::cout << num << std::endl; // prints 15
11 }
```

# Function Call using Variable Address

- C++ allows call-by-value using pointer **int \*x**

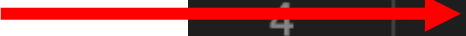


```
1  #include <iostream>
2
3  void addTen(int *x) {
4      *x = *x + 10;  // dereference pointer to change original
5  }
6
7  int main() {
8      int num = 5;
9      addTen(&num);  // pass address
10     std::cout << num;  // prints 15
11 }
```

# Call-by-Value using Variable Address

- Need to de-reference the address to modify variable

```
1  #include <iostream>
2
3  void addTen(int *x) {
4      *x = *x + 10;  // dereference pointer to change original
5  }
6
7  int main() {
8      int num = 5;
9      addTen(&num);  // pass address
10     std::cout << num;  // prints 15
11 }
```





# Examples – What is the output?

```
1  #include <stdio>
2
3  void main () {
4      int a = 24601;
5      int& a_ref = a;
6      int* a_ptr = &a;
7
8      printf("%d\n", a);
9      printf("%d\n", a_ref);
10     printf("%p\n", a_ptr);
11 }
```

# Examples – What is the output?

```
1  #include <stdio>
2
3  void main () {
4      int a = 24601;
5      int& a_ref = a;
6      int* a_ptr = &a;
7
8      printf("%d\n", a); //24601
9      printf("%d\n", a_ref);
10     printf("%p\n", a_ptr);
11 }
```

# Examples – What is the output?

```
1  #include <stdio>
2
3  void main () {
4      int a = 24601;
5      int& a_ref = a;
6      int* a_ptr = &a;
7
8      printf("%d\n", a); //24601
9      printf("%d\n", a_ref); //24601
10     printf("%p\n", a_ptr);
11 }
```

# Examples – What is the output?

```
1  #include <stdio>
2
3  void main () {
4      int a = 24601;
5      int& a_ref = a;
6      int* a_ptr = &a;
7
8      printf("%d\n", a); //24601
9      printf("%d\n", a_ref); //24601
10     printf("%p\n", a_ptr); //0x7fffffffddcdc
11 }
```

# Examples – What is the output?

```
1 int a = 24601;
2 int b = 24602;
3 int& a_ref = a;
4 int* a_ptr = &a;
5
6 printf("%d\n", a); //24601
7 printf("%d\n", a_ref); //24601
8 printf("%p\n", a_ptr); //0x7fffffffddcc8
9
10 a_ref = b; // assign b to a_ref
11
12 printf("%d\n", a_ref);
13 printf("%d\n", a);
14 printf("%p\n", &b);
15 printf("%p\n", &a_ref);
```

# Examples – What is the output?

```
1 int a = 24601;
2 int b = 24602;
3 int& a_ref = a;
4 int* a_ptr = &a;
5
6 printf("%d\n", a); //24601
7 printf("%d\n", a_ref); //24601
8 printf("%p\n", a_ptr); //0x7fffffffddcc8
9
10 a_ref = b; // assign b to a_ref
11
12 printf("%d\n", a_ref);
13 printf("%d\n", a);
14 printf("%p\n", &b);
15 printf("%p\n", &a_ref);
```

# Examples – References & Pointers

```
1  int a = 24601;
2  int b = 24602;
3  int& a_ref = a;
4  int* a_ptr = &a;
5
6  printf("%d\n", a); //24601
7  printf("%d\n", a_ref); //24601
8  printf("%p\n", a_ptr); //0x7fffffffddcc8
9
10 a_ref = b; // assign b to a_ref
11
12 printf("%d\n", a_ref);
13 printf("%d\n", a);
14 printf("%p\n", &b);
15 printf("%p\n", &a_ref);
```

# Examples – References & Pointers

```
1 int a = 24601;
2 int b = 24602;
3 int& a_ref = a;
4 int* a_ptr = &a;
5
6 printf("%d\n", a); //24601
7 printf("%d\n", a_ref); //24601
8 printf("%p\n", a_ptr); //0x7fffffffddcc8
9
10 a_ref = b; // assign b to a_ref
11
12 printf("%d\n", a_ref); //24602 <- value of the reference has changed
13 printf("%d\n", a); //24602 <- value of a has changed
14 printf("%p\n", &b); //0x7fffffffddcc4
15 printf("%p\n", &a_ref); //0x7fffffffddcc8 <- a_ref still is a reference to a
```



# Code Organization

- Separation into
  - Header (.hpp)
  - Source (.cpp)
- Header
  - Class definition
  - Method declaration
- Source
  - Method implementation (definition)

# Example

## Time.hpp

```
class Time
{
public:
    Time(int hour, int minutes);
    void displayTime();

private:
    int hour_;
    int minute_;
}
```

## Time.cpp

```
#include "time.hpp"

Time::Time(int hour, int minute) :
    hour_(hour), minute_(minute)
{
}

void Time::displayTime()
{
    printf("It is %d:%d!", hour_, minute_);
}
```

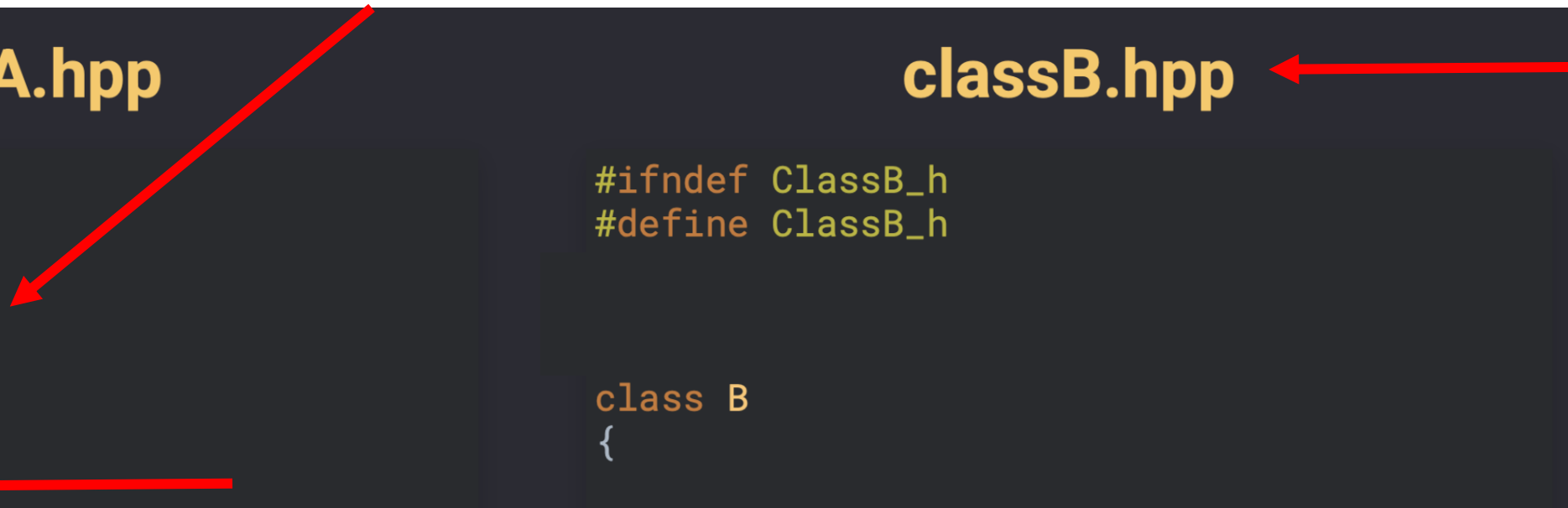
# Dependencies

## classA.hpp

```
#ifndef ClassA_h
#define ClassA_h

#include "classB.hpp"

class A
{
    B* oldB_;
public:
    A(B* oldB) : oldB_(oldB){}
    void run() { if(oldB_) oldB_->run(); }
};
#endif
```



## classB.hpp

```
#ifndef ClassB_h
#define ClassB_h

class B
{
public:
    void run() { }
};
#endif
```

# Dependencies

- Circular dependency!

## classA.hpp

```
#ifndef ClassA_h
#define ClassA_h

#include "classB.hpp"

class A
{
    B* oldB_;
public:
    A(B* oldB) : oldB_(oldB){}
    void run() { if(oldB_) oldB_->run(); }
};
#endif
```

## classB.hpp

```
#ifndef ClassB_h
#define ClassB_h

#include "classA.hpp" ←

class B
{
    A* neA_; ←
public:
    B(A* neA) : neA_(neA){}
    void run() { if(neA_) neA_->run(); }
};
#endif
```

# Forward Declaration

- To just tell the compiler “it exists,” without including all details yet
- Can **only use a pointer or reference**
- No #include needed

## classA.hpp

```
#ifndef ClassA_h
#define ClassA_h
// there is a class B, you
// don't need to know how it looks like
class B; ←
class A
{
    B* oldB_;
public:
    A(B* oldB) : oldB_(oldB){}
    void run();
};
#endif
```

## classB.hpp

```
#ifndef ClassB_h
#define ClassB_h
// there is a class A, you
// don't need to know how it looks like
class A; ←
class B
{
    A* neA_;
public:
    B(A* neA) : neA_(neA){}
    void run();
};
#endif
```

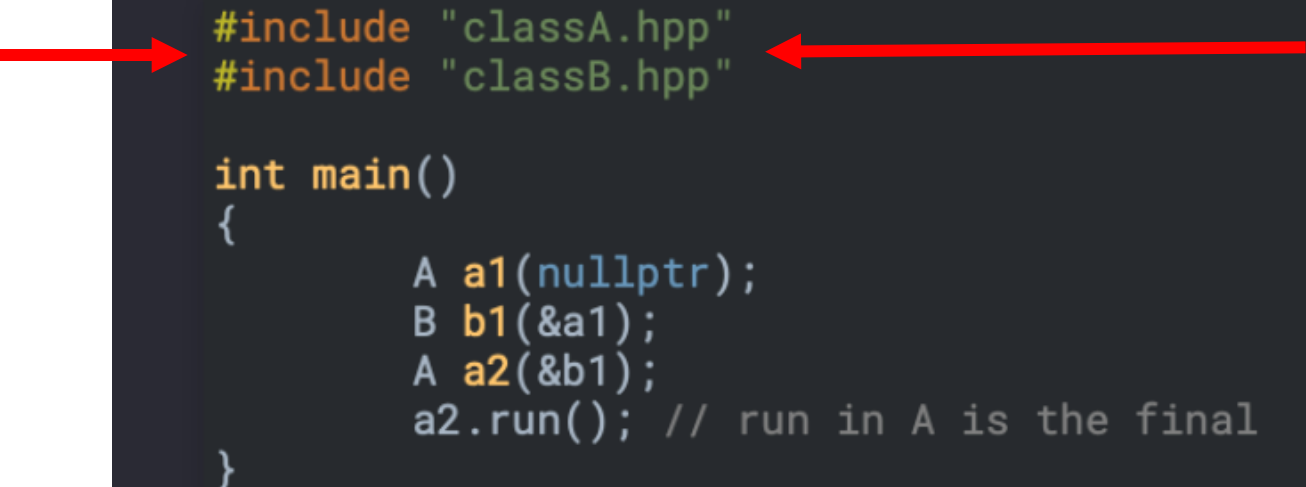
# Include when actually needed to create object

## classA.cpp

```
#include "classA.hpp"
#include "classB.hpp"
// now we know how both A and B look like!
void A::run()
{
    if(oldB_) oldB_>run();
    else printf("run in A is the final\n");
}
```

## classB.cpp

```
#include "classB.hpp"
#include "classA.hpp"
// now we know how both B and A look like!
void B::run()
{
    if(newA_) newA_>run();
    else printf("run in B is the final\n");
}
```



```
#include "classA.hpp"
#include "classB.hpp"

int main()
{
    A a1(nullptr);
    B b1(&a1);
    A a2(&b1);
    a2.run(); // run in A is the final
}
```

