Object-Oriented Programming

Reference Types III & Object Lifecycle

ReCap - const

- Compiler prohibits modifications

```
1   int main()
2   {
3       int counter{ 0 };
4       counter++; // OK
5
6       const int max_limit{ counter + 19 }; // initialize constant to 20
7       max_limit++; // error: cannot assign to variable
8                    // with const-qualified type 'const int'
9
10      return 0;
11  }
```

ReCap - const

- Constants must be initialized upon definition

```
const int minutesPerHour = 60; // defined and initialized
```

ReCap - const

- Constants must be initialized upon definition

```
const int minutesPerHour = 60; // defined and initialized
```

➔ERROR

```
const int minutesPerHour;
minutesPerHour = 60;
```

ReCap - const

- Constants must be initialized upon definition

```
const int minutesPerHour = 60; // defined and initialized
```

➔ERROR

```
const int minutesPerHour;
minutesPerHour = 60;
```

Because a const variable cannot change after it is created

ReCap - **const-Attributes**

- Constants must be initialized upon definition
  - In header

```
class Time {
    const int offset = 42;
};
```

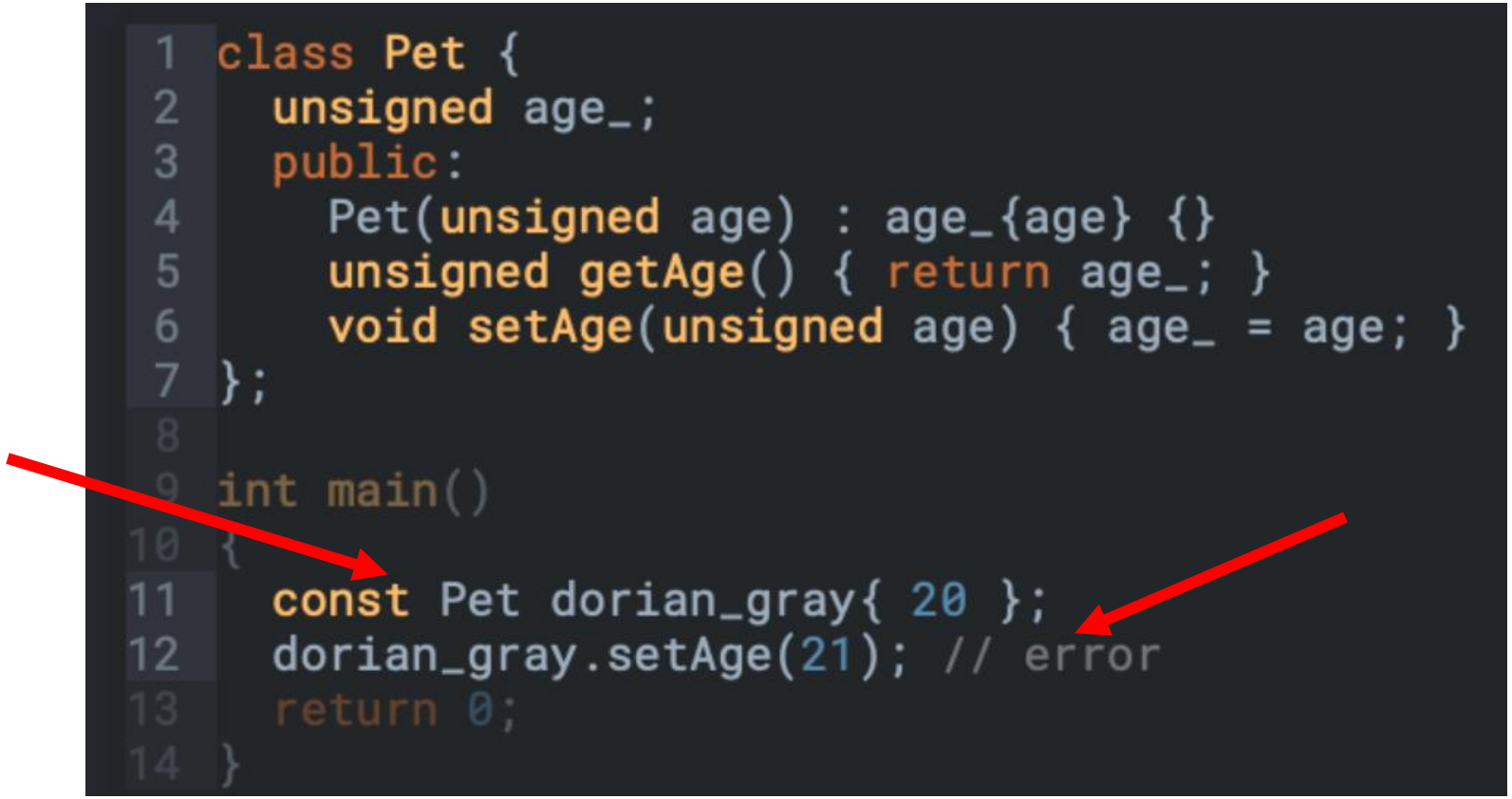
  - Using initializer list

```
Time(int off) { offset = off; } // ERROR: Forbidden!
```

```
class Time {
    const int offset;
public:
    Time(int off) : offset(off) {}
};
```

Const Objects

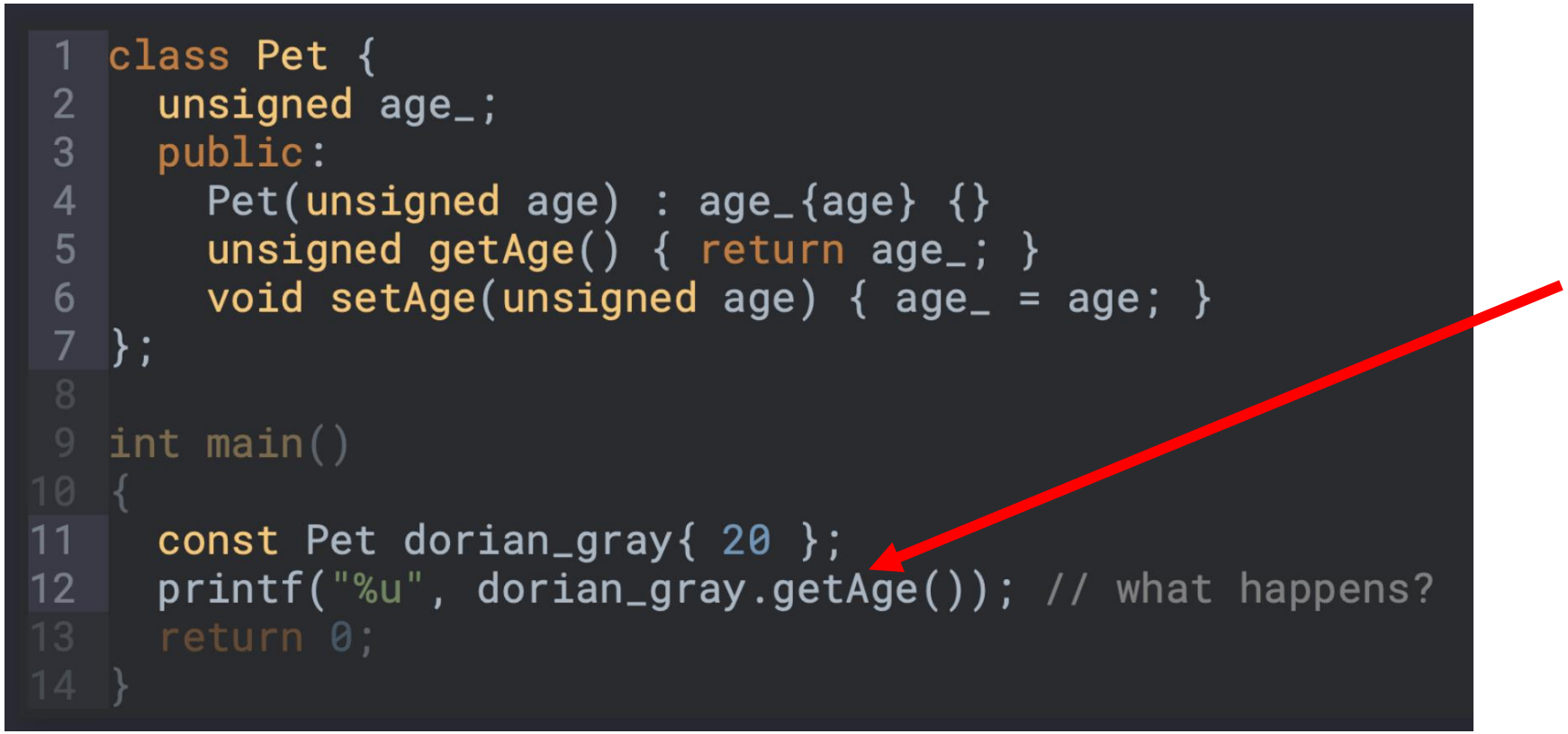
- Compiler prohibits modification of non-static attributes in const objects.



```
1  class Pet {
2    unsigned age_;
3    public:
4      Pet(unsigned age) : age_{age} {}
5      unsigned getAge() { return age_; }
6      void setAge(unsigned age) { age_ = age; }
7  };
8
9  int main()
10 {
11   const Pet dorian_gray{ 20 };
12   dorian_gray.setAge(21); // error
13   return 0;
14 }
```

Const Objects

- non-const member functions cannot be called on const objects



```
1  class Pet {
2    unsigned age_;
3    public:
4      Pet(unsigned age) : age_{age} {}
5      unsigned getAge() { return age_; }
6      void setAge(unsigned age) { age_ = age; }
7  };
8
9  int main()
10 {
11   const Pet dorian_gray{ 20 };
12   printf("%u", dorian_gray.getAge()); // what happens?
13   return 0;
14 }
```

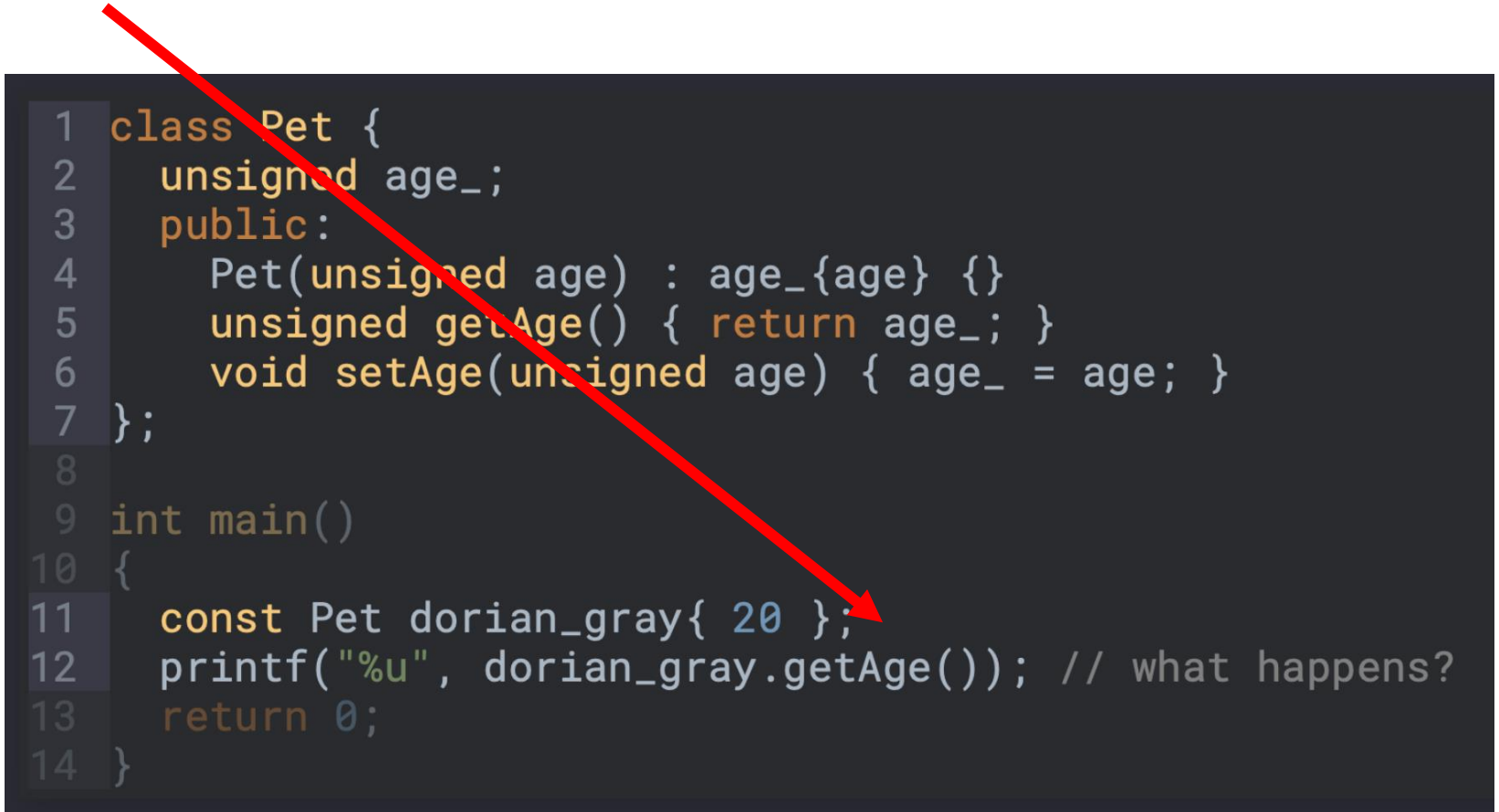
Const Objects

- non-const member functions cannot be called on const objects
- **const** object, which forbids modification (even potential modification)

```
1  class Pet {
2    unsigned age_;
3    public:
4      Pet(unsigned age) : age_{age} {}
5      unsigned getAge() { return age_; }
6      void setAge(unsigned age) { age_ = age; }
7  };
8
9  int main()
10 {
11   const Pet dorian_gray{ 20 };
12   printf("%u", dorian_gray.getAge()); // what happens?
13   return 0;
14 }
```

Const Objects

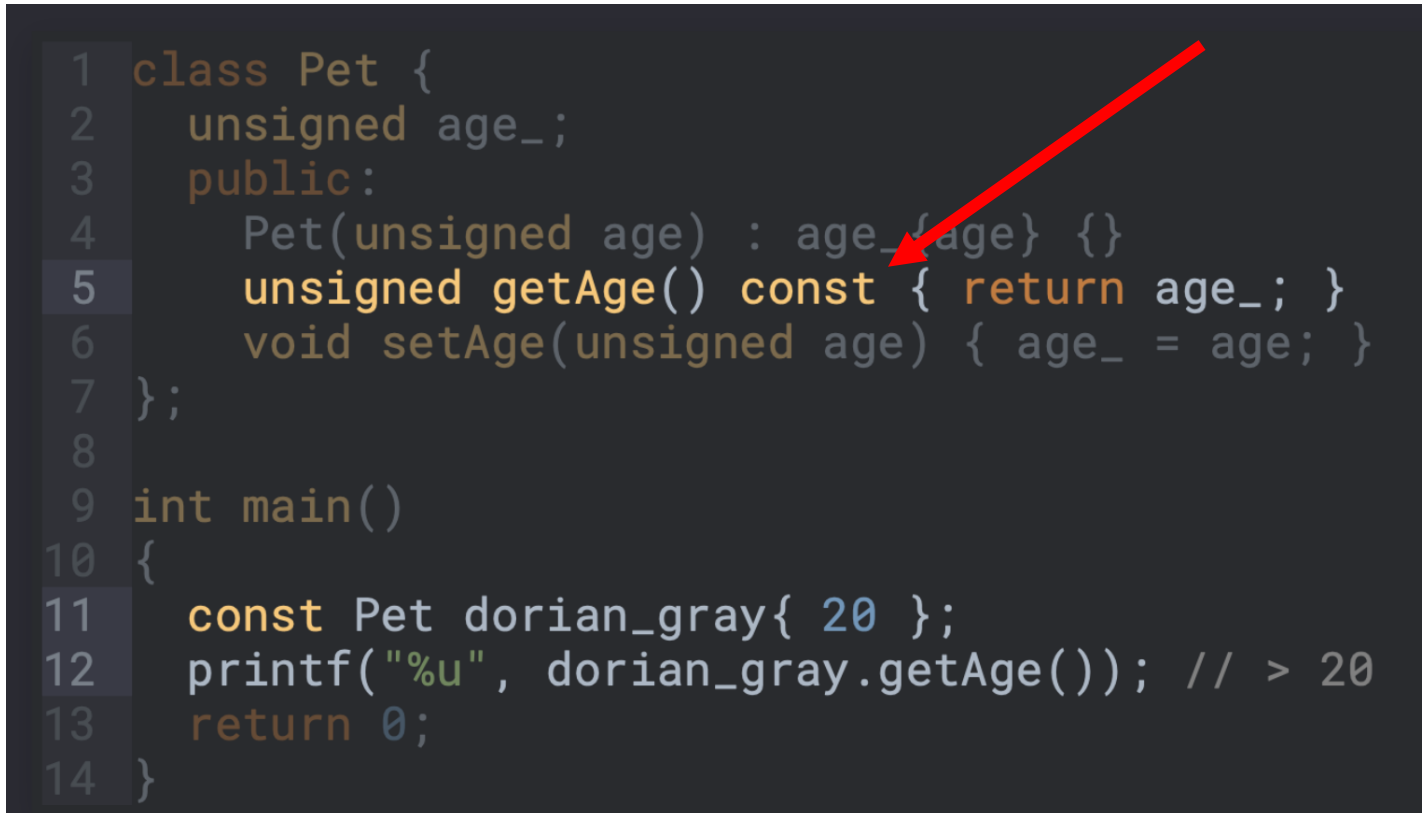
- A const object can only call **const methods**



```
1  class Pet {
2    unsigned age_;
3    public:
4      Pet(unsigned age) : age_{age} {}
5      unsigned getAge() { return age_; }
6      void setAge(unsigned age) { age_ = age; }
7  };
8
9  int main()
10 {
11   const Pet dorian_gray{ 20 };
12   printf("%u", dorian_gray.getAge()); // what happens?
13   return 0;
14 }
```

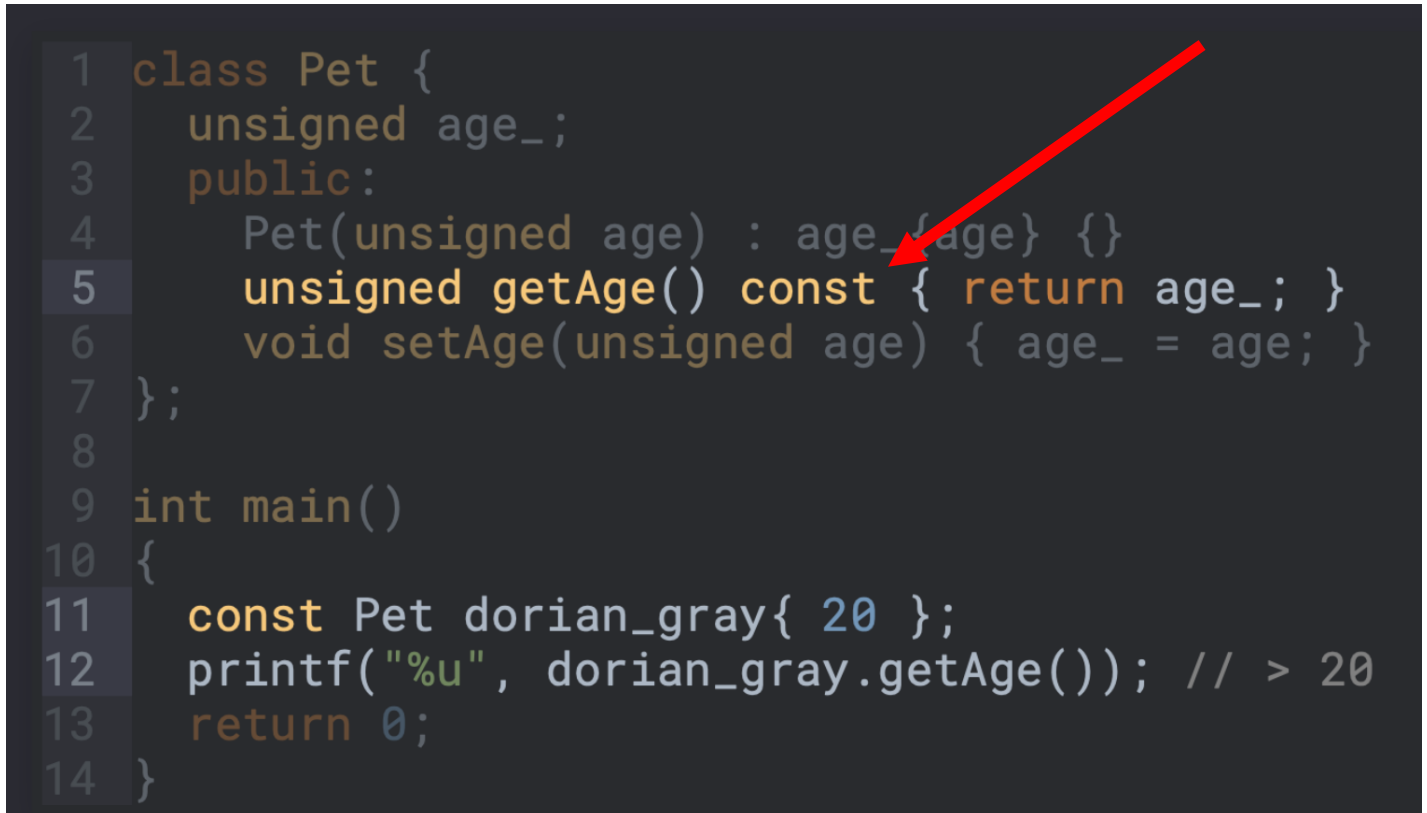
const Methods

- const **after parentheses** declares this as a **const method**



const Methods

- Object is considered read-only inside this function



const Methods

- Allows the method to be called for a constant object
- Compiler prohibits modification of (non-static) attributes

const Methods

- Good code: Always use const if method does not change the object
  For example: for Getter methods

const Methods

- Good code: Always use const if method does not change the object
  For example: for Getter methods
  - Prevents accidental modification of object state

const Methods

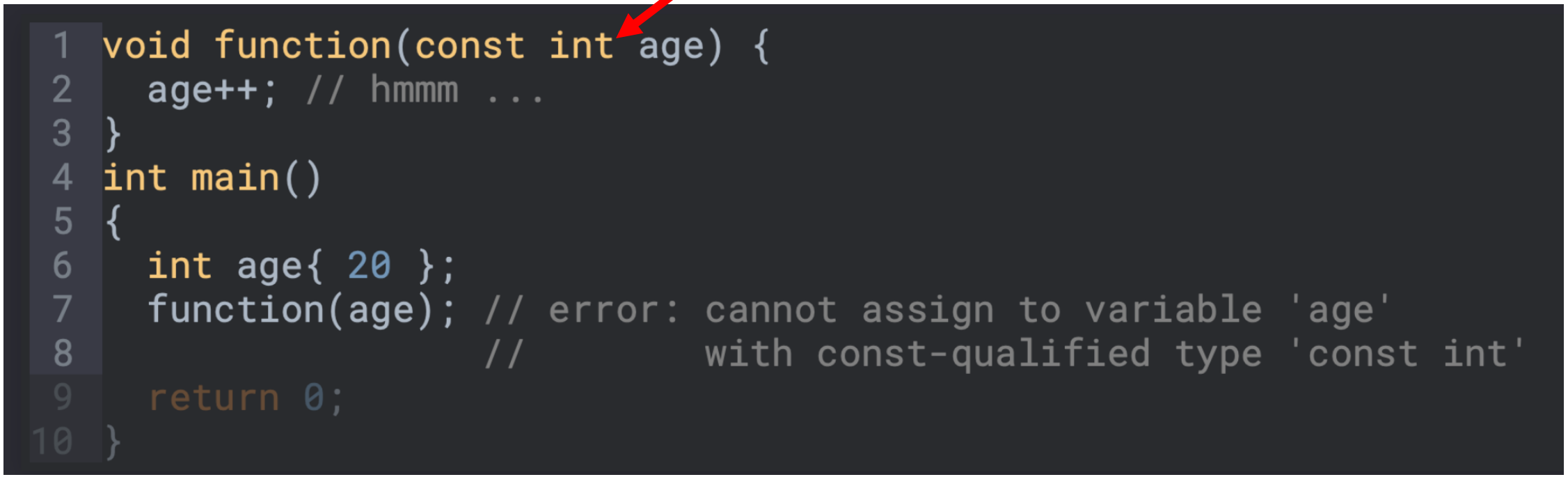
- Good code: Always use const if method does not change the object
  For example: for Getter methods
  - Prevents accidental modification of object state
  - Signals to users' which methods are read-only

const Methods

- Good code: Always use const if method does not change the object
  For example: for Getter methods
  - Prevents accidental modification of object state
  - Signals to users' which methods are read-only
  - Increases flexibility: can be called on const objects

const function parameters

- Compiler prohibits modification in the function body
- Guarantees that the passed value will not be changed.



```
1  void function(const int age) {
2    age++; // hmmm ...
3  }
4  int main()
5  {
6    int age{ 20 };
7    function(age); // error: cannot assign to variable 'age'
8                   //        with const-qualified type 'const int'
9    return 0;
10 }
```

const function parameters

- Compiler prohibits modification in the function body
- Guarantees that the passed value will not be changed.
  - Important for call-by-reference or call-by-value using pointer

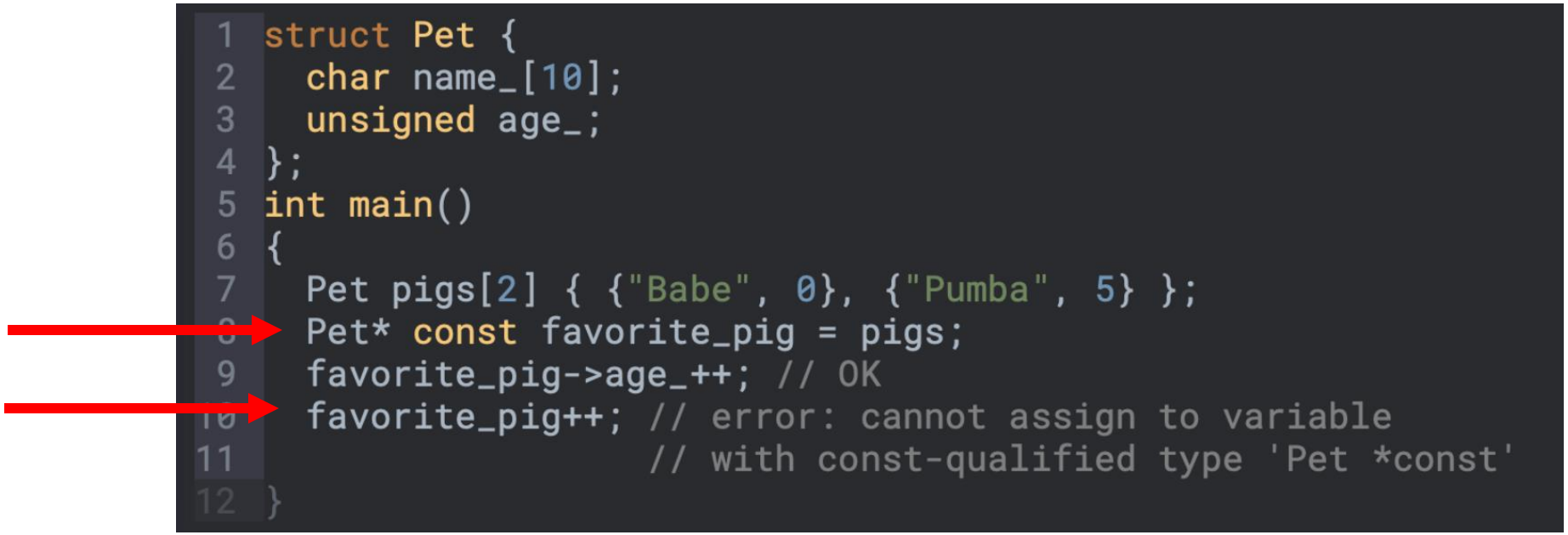
const function parameters

- const parameters change the signature

```
1  // two different functions:
2  void happyBirthday(const Pet& pet)
3  {
4    printf("happy birthday!");
5  }
6
7  void happyBirthday(Pet& pet)
8  {
9    printf("happy birthday!");
10   pet.setAge( pet.getAge() + 1 );
11 }
```

const pointers

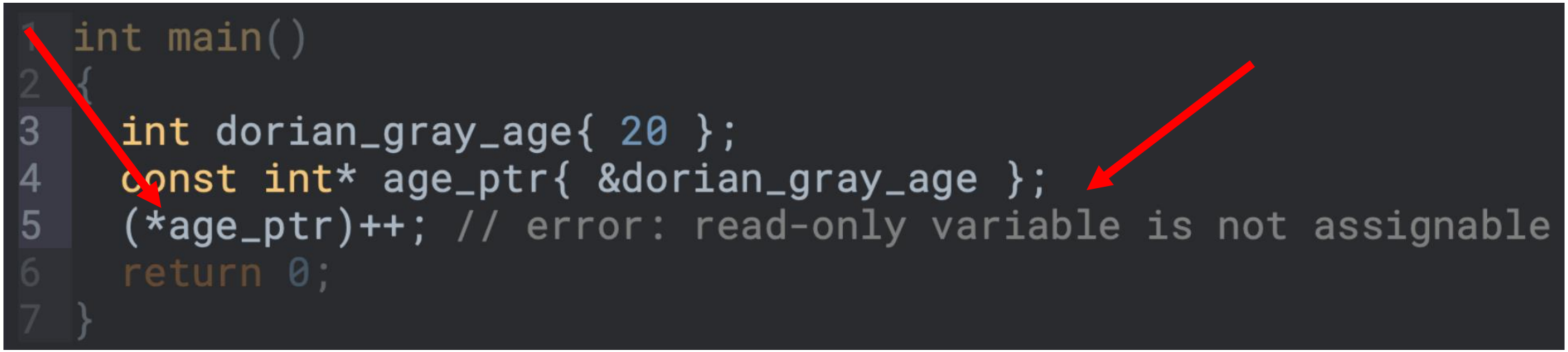
- Compiler prohibits modification **of the pointer**
- Cannot increment integer



```
1  struct Pet {
2    char name_[10];
3    unsigned age_;
4  };
5  int main()
6  {
7    Pet pigs[2] { {"Babe", 0}, {"Pumba", 5} };
8    Pet* const favorite_pig = pigs;
9    favorite_pig->age_++; // OK
10   favorite_pig++; // error: cannot assign to variable
11                   // with const-qualified type 'Pet *const'
12 }
```

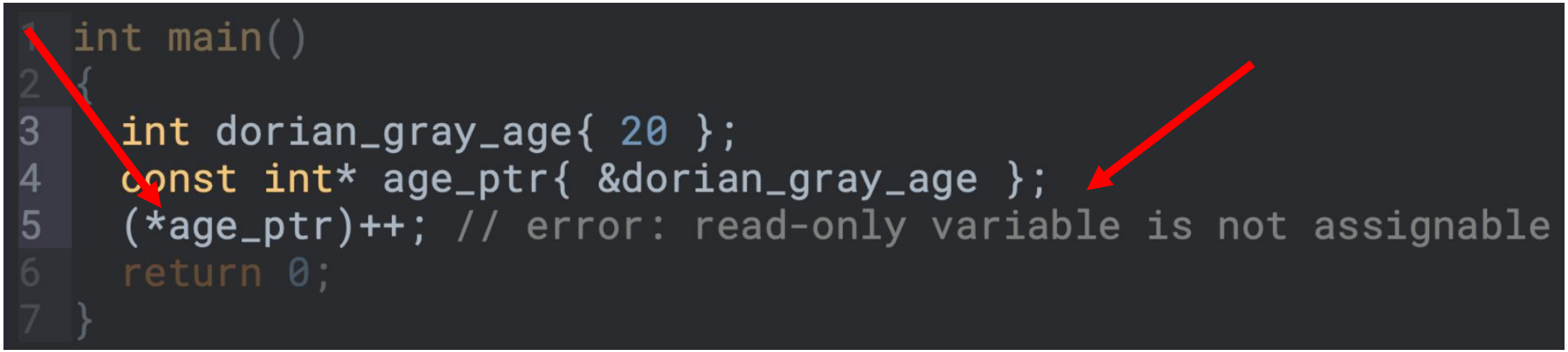
const pointers

- Cannot increment
- Compiler prohibits modification **of the integer**



const pointers

- Note: const int * is equal to int const *
  - const always refers to the previous token, if there is none, then it refers to the next token.

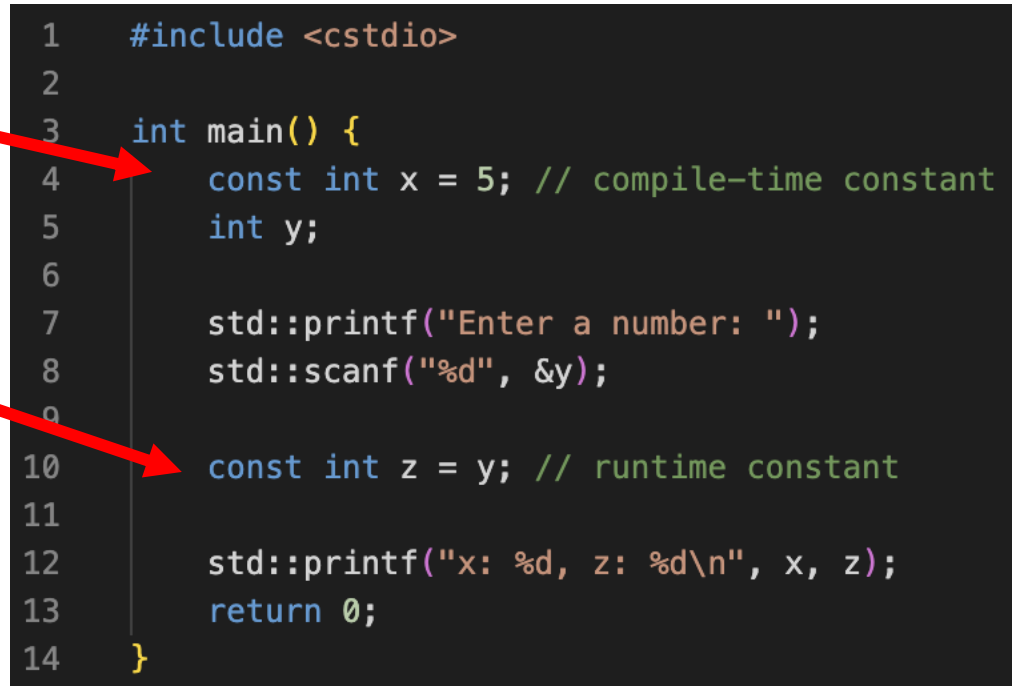


const vs constexpr

- Const
  - Declares a variable whose value cannot be changed after initialization.

const vs constexpr

- Const
  - Declares a variable whose value cannot be changed after initialization.
  - Compile-time constant
  - Run-time constant
    - the compiler doesn't know its value



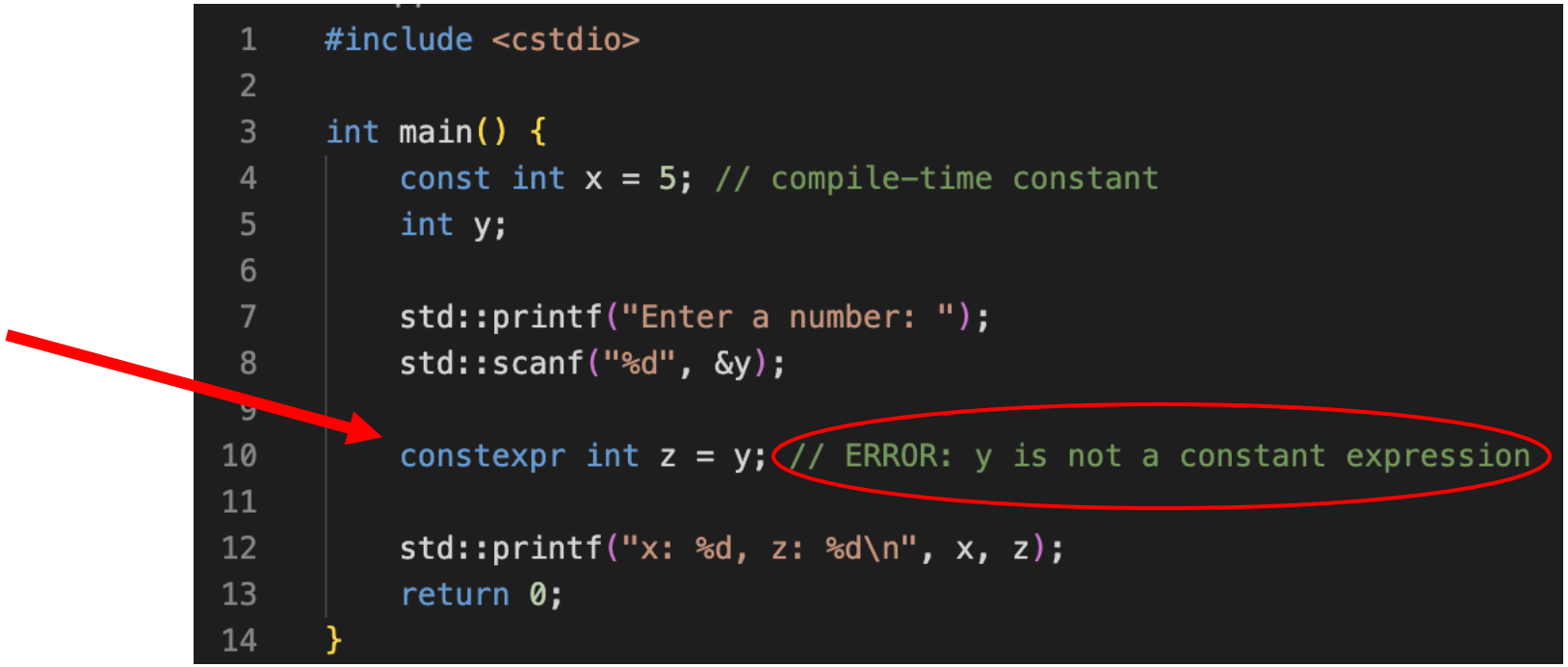
```cpp
#include <cstdio>

int main() {
    const int x = 5; // compile-time constant
    int y;

    std::printf("Enter a number: ");
    std::scanf("%d", &y);

    const int z = y; // runtime constant

    std::printf("x: %d, z: %d\n", x, z);
    return 0;
}
```

const vs constexpr

- Constexpr
  - Forces compile-time evaluation

const vs constexpr

- Constexpr
  - Forces compile-time evaluation



```cpp
#include <cstdio>

int main() {
    const int x = 5; // compile-time constant
    int y;

    std::printf("Enter a number: ");
    std::scanf("%d", &y);

    constexpr int z = y; // ERROR: y is not a constant expression

    std::printf("x: %d, z: %d\n", x, z);
    return 0;
}
```

const vs constexpr

- Constexpr
  - Forces compile-time evaluation
  - Can be used with functions that can be evaluated at compile time

```cpp
#include <iostream>

constexpr int square(int n) {
    return n * n;
}

int main() {
    constexpr int a = 5;           // known at compile time
    constexpr int b = square(a);   // also compile-time

    std::cout << "b: " << b << std::endl;

    int arr[b]; // valid: array size must be compile-time constant

    return 0;
}
```

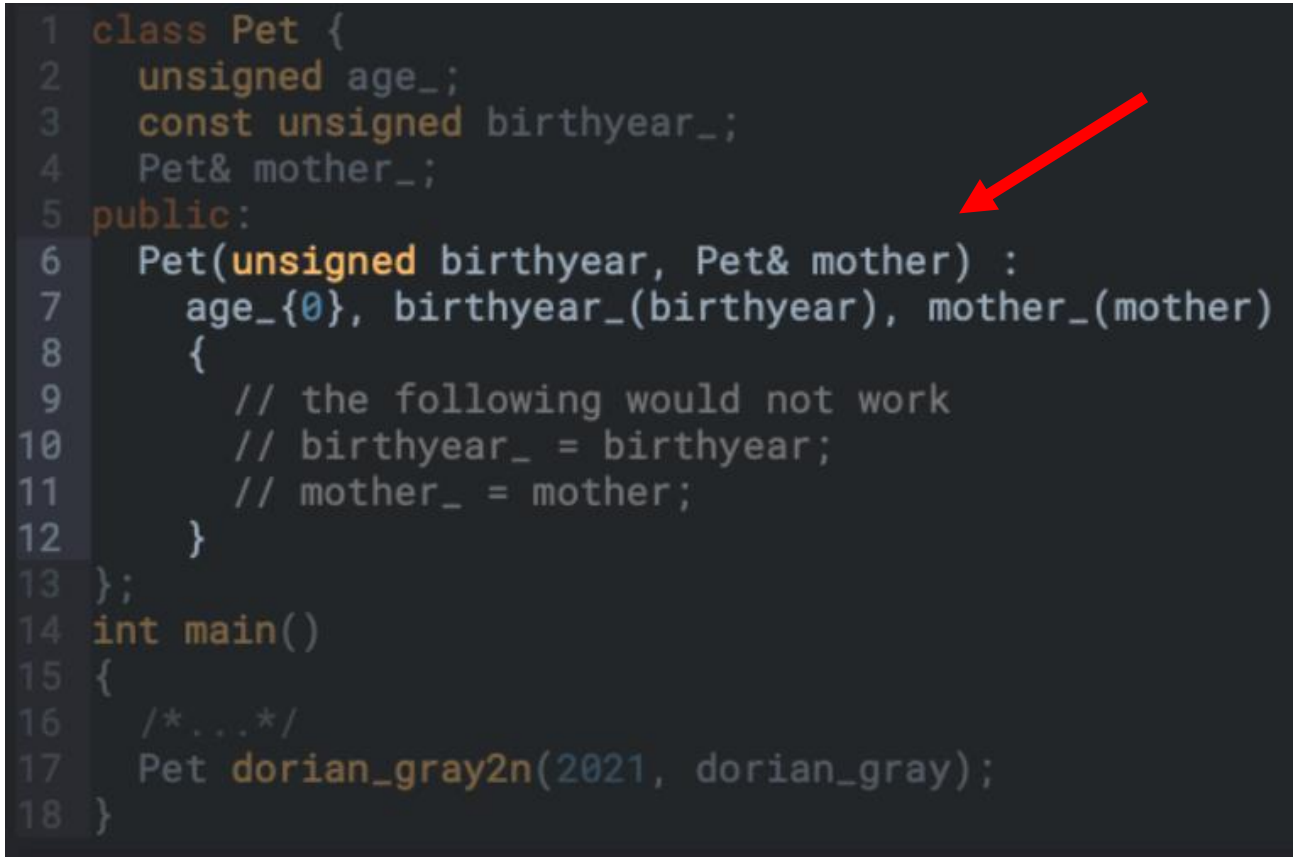
Initialization using initialization list

- const members must be initialized in the initialization list.

```
class Pet {
  unsigned age_;
  const unsigned birthyear_;
  Pet& mother_;
public:
  Pet(unsigned birthyear, Pet& mother) :
    age_{0}, birthyear_(birthyear), mother_(mother)
    {
      // the following would not work
      // birthyear_ = birthyear;
      // mother_ = mother;
    }
};
int main()
{
  /*...*/
  Pet dorian_gray2n(2021, dorian_gray);
}
```

Initialization using initialization list

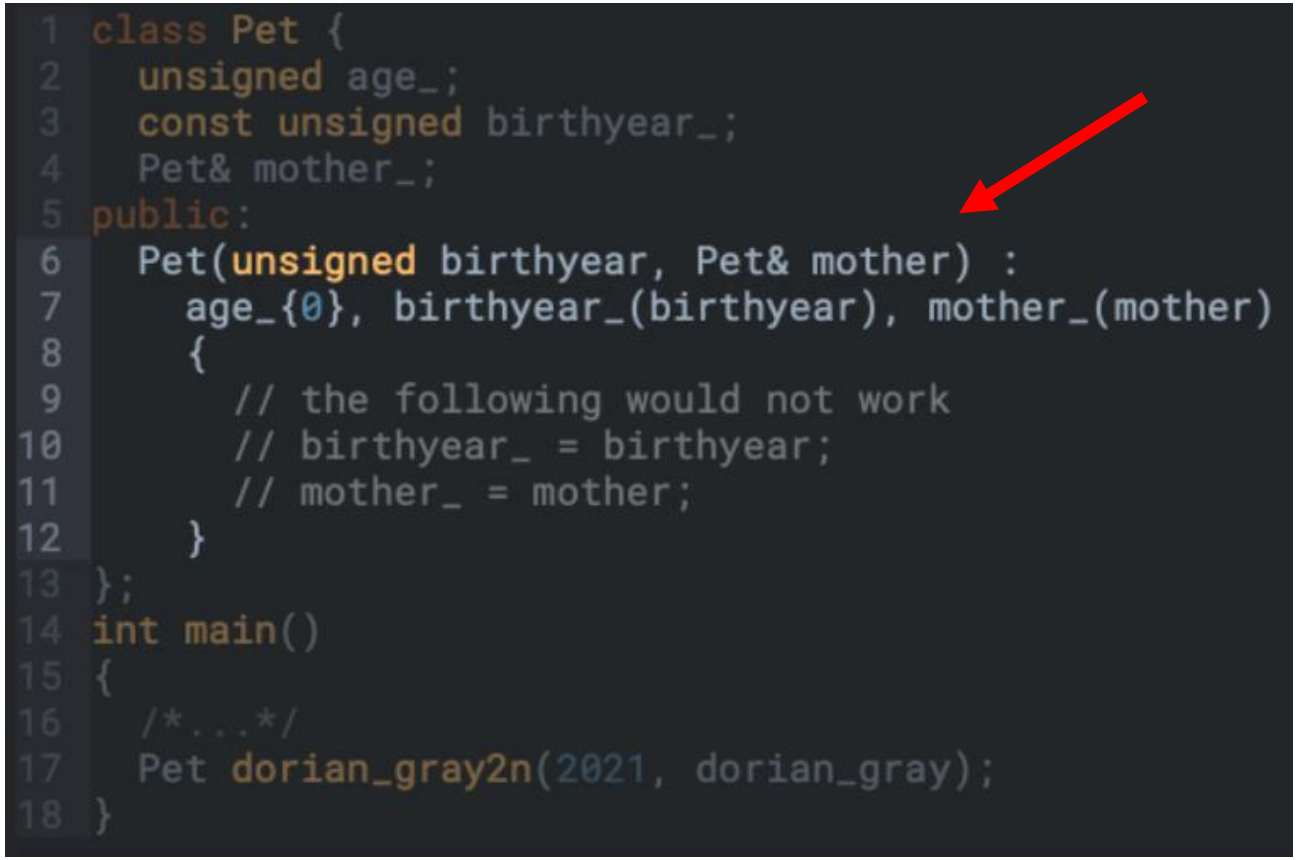
- Reference members must also be initialized in the initialization list.



```
class Pet {
  unsigned age_;
  const unsigned birthyear_;
  Pet& mother_;
public:
  Pet(unsigned birthyear, Pet& mother) :
    age_{0}, birthyear_(birthyear), mother_(mother)
    {
      // the following would not work
      // birthyear_ = birthyear;
      // mother_ = mother;
    }
};
int main()
{
  /*...*/
  Pet dorian_gray2n(2021, dorian_gray);
}
```

Initialization using initialization list

- Reference members must also be initialized in the initialization list.
- Reference must always be bound to a valid object



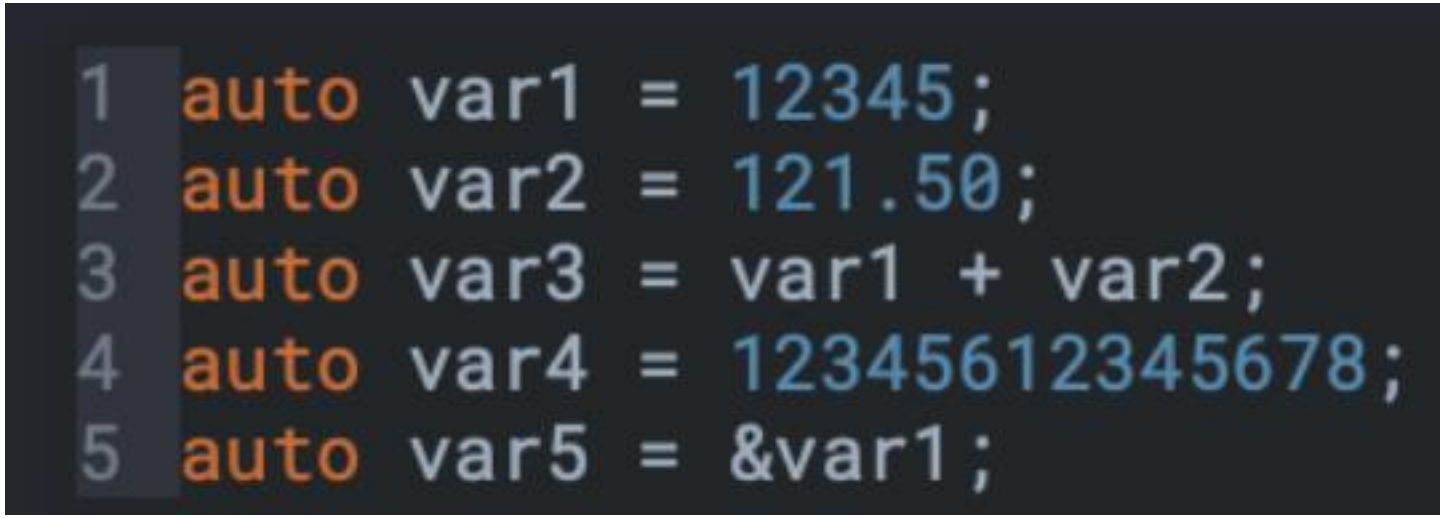
```
class Pet {
  unsigned age_;
  const unsigned birthyear_;
  Pet& mother_;
public:
  Pet(unsigned birthyear, Pet& mother) :
    age_{0}, birthyear_(birthyear), mother_(mother)
    {
      // the following would not work
      // birthyear_ = birthyear;
      // mother_ = mother;
    }
};
int main()
{
  /*...*/
  Pet dorian_gray2n(2021, dorian_gray);
}
```

Data type

- Use auto to leave decision on data type assignment to compiler.

auto example

- Which data types are automatically assigned?



auto example

- Which data types are automatically assigned?

```
1 auto var1 = 12345;              //Integer
2 auto var2 = 121.50;             //Double
3 auto var3 = var1 + var2;        //Double
4 auto var4 = 123456123456678;    //Long
5 auto var5 = &var1;              //Pointer to Integer
```

auto

- Then, I simply use auto everywhere!!

auto

- Then, I simply use auto everywhere!! ...not the best idea!

- auto makes the code difficult to parse by humans.

auto

- Then, I simply use auto everywhere!! ...not the best idea!

- auto makes the code difficult to parse by humans.
- auto leads to errors because it is no longer easy to see what is happening.

auto

- Then, I simply use auto everywhere!! ...not the best idea!

- auto makes the code difficult to parse by humans.
- auto leads to errors because it is no longer easy to see what is happening.
- auto can lead to unwanted data types.

auto

- Then, I simply use auto everywhere!! ...not the best idea!

- auto makes the code difficult to parse by humans.
- auto leads to errors because it is no longer easy to see what is happening.
- auto can lead to unwanted data types.
- auto leads to references and pointers being forgotten

auto

- Use auto...
  - ... if writing out the type reduces readability.

```
std::map<CompressedBirthday<unsigned long long>, std::pair<int, int>>::iterator iterator mymap_.begin();
auto iterator = mymap_.begin();
```

auto

- Use auto...
  - ... if writing out the type reduces readability.

```
std::map<CompressedBirthday<unsigned long long>, std::pair<int, int>>::iterator iterator mymap_.begin();
auto iterator = mymap_.begin();
```

  - ... if the data type could be easily exchanged.
    e.g. map from above is replaced by unordered_map

auto

- Use auto...
  - ... if writing out the type reduces readability.

```
std::map<CompressedBirthday<unsigned long long>, std::pair<int, int>>::iterator iterator mymap_.begin();
auto iterator = mymap_.begin();
```

  - ... if the data type could be easily exchanged.
    e.g. map from above is replaced by unordered_map

  - ... if the data type has not yet been determined when the code is written.
    e.g. **templates** (we will cover them later)

Loops

- Traditional

```c
int main() {
    char arr[] = {'H', 'a', '1', '1', '0'};
    for (int i = 0; i < 5; ++i)
    {
        char el = arr[i];
        printf("%c", el);
    }
}
```

Loops in C++

- Range-based

```
int main() {
    char arr[] = {'H', 'a', '1', '1', '0'};
    for (char el : arr) {
        printf("%c", el);
    }
}
```

Range based Loops with auto

- Output?

```
int main() {
    int arr[] = {1, 2, 3, 27, 42};
    for (auto& el : arr) {
        printf("%d", el);
    }
}
```

Range based Loops with auto and modifier

- Output?

```
int main() {
    int arr[] = {1, 2, 3, 27, 42};
    for (auto el : arr) {
        printf("%d", ++el);
    }
}
```

Range based Loops with auto, reference and modifier

- Output?

```
int main() {
    int arr[] = {1, 2, 3, 27, 42};
    for (auto& el : arr) {
        printf("%d", ++el);
    }
}
```

Range based Loops with auto, reference and modifier and const

- Output?

```
int main() {
    int arr[] = {1, 2, 3, 27, 42};
    for (const auto el : arr) {
        printf("%d", ++el);
    }
}
```

Range based Loops with auto, reference and modifier and const

- Output?

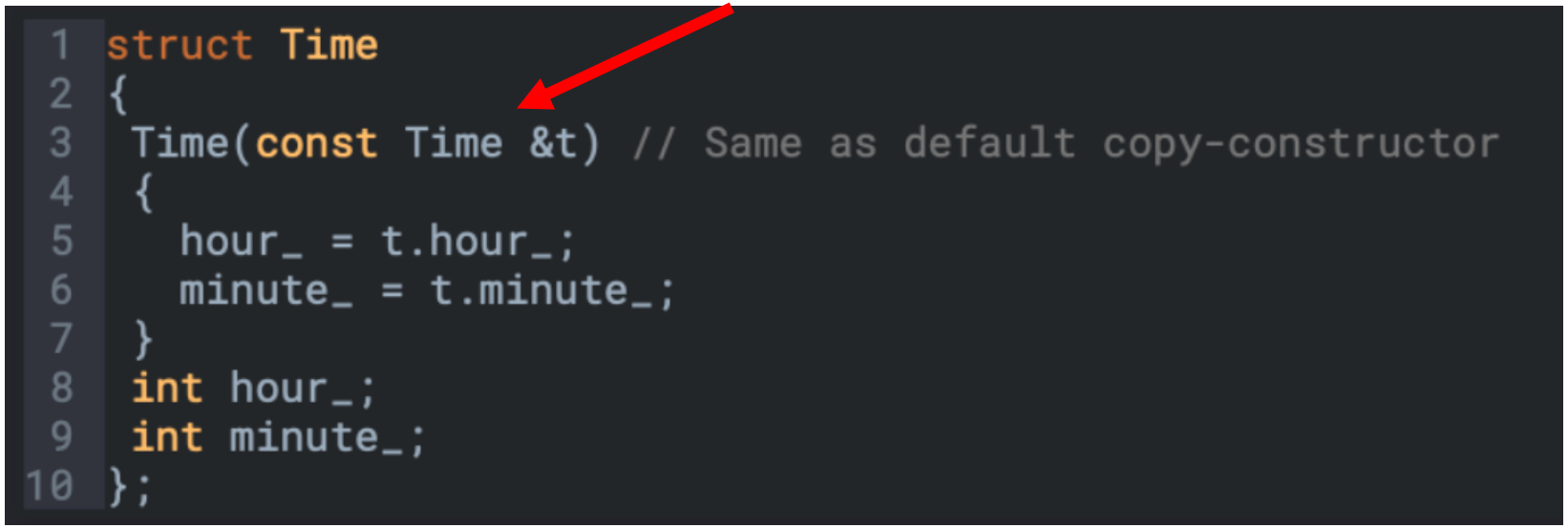
```
int main() {
    int arr[] = {1, 2, 3, 27, 42};
    for (const auto el : arr) {
        printf("%d", ++el);
    }
}
```

// error : cannot increment variable 'el'
// with const-qualified type 'const int &'

Object Copy

- The copy constructor is a special constructor that creates an object by initializing it with another object (of the same class)

Copy Constructor



```
1  struct Time
2  {
3   Time(const Time &t) // Same as default copy-constructor
4   {
5     hour_ = t.hour_;
6     minute_ = t.minute_;
7   }
8   int hour_;
9   int minute_;
10 };
```

Object Copy

- When is a copy constructor called?
  - When an object is initialized with an object of the same class.

Object Copy

- When is a copy constructor called?
  - When an object is initialized with an object of the same class.
  - When an object is passed by value (call-by-value).

Object Copy

- When is a copy constructor called?
  - When an object is initialized with an object of the same class.
  - When an object is passed by value (call-by-value).
  - When an object with the same type is returned.

```
1    Person makePerson() {
2        Person temp(40);
3        return temp;   // Copy constructor used
4    }
5
6    int main() {
7        Person p = makePerson();
8    }
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};
    Time t2(t1);
    Time t3{t1};
    Time t4 = t1;
    Time t5 = addHour(t1);
    printTimeRef(t1);
}
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};       //Regular constructor
    Time t2(t1);
    Time t3{t1};
    Time t4 = t1;
    Time t5 = addHour(t1);
    printTimeRef(t1);
}
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};        //Regular constructor
    Time t2(t1);            //Copy-constructor
    Time t3{t1};
    Time t4 = t1;
    Time t5 = addHour(t1);
    printTimeRef(t1);
}
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};        //Regular constructor
    Time t2(t1);            //Copy-constructor
    Time t3{t1};            //Copy-constructor
    Time t4 = t1;           //Copy-constructor
    Time t5 = addHour(t1);
    printTimeRef(t1);
}
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};        //Regular constructor
    Time t2(t1);            //Copy-constructor
    Time t3{t1};            //Copy-constructor
    Time t4 = t1;           //Copy-constructor
    Time t5 = addHour(t1); // Copy-constructor (call by value)
    printTimeRef(t1);
}
```

Example

```
class Time
{
private:
    int hour_;
    int minute_;

public:
    Time(){ printf("Constructor is called\n"); }
    Time(const Time &t) {
        printf("Copy-Constructor is called\n");
        hour_ = t.hour_;
        minute_ = t.minute_;
    }
};

Time addHour(Time time)
{
    time.hour_++;
    return time; // Copy constructor (return) *not always*
}

void printTimeRef(const Time& time)
{
    std::cout << std::format("{}:{}\n", time.hour_, time.minute_);
}

int main()
{
    Time t1{17, 40};        //Regular constructor
    Time t2(t1);            //Copy-constructor
    Time t3{t1};            //Copy-constructor
    Time t4 = t1;           //Copy-constructor
    Time t5 = addHour(t1); // Copy-constructor (call by value)
    printTimeRef(t1);       // *No* Copy-constructor (call by reference)
}
```

Destructor (Delete Object)

- The destructor is called automatically when an instance of a class is destroyed.

```
1  class Time
2  {
3  public:
4      Time(){ printf("constructor called\n"); }
5      ~Time(){ printf("destructor called\n"); }
6  };
7
8  int main()
9  {
10     Time t2;  // constructor called
11     return 0; // destructor called
12 }
```

Destructor (Delete Object)

- The destructor is called automatically when an instance of a class is destroyed.

Scope

- **Scope** in C++ defines the region of a program where an identifier (a variable, function, class, etc.) is visible and can be accessed by its name

- Example: Two variables named count in different scopes are completely independent

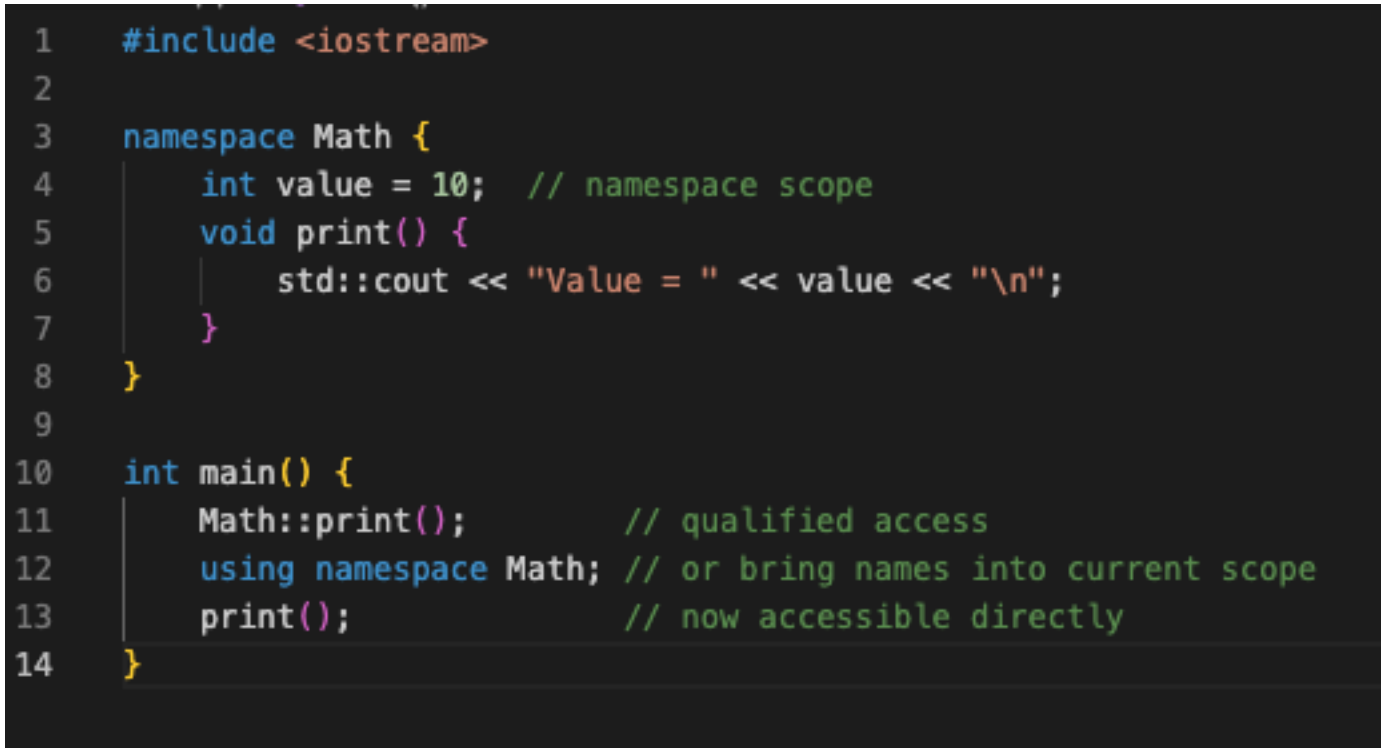
Types of Scopes

- Global scope
  - accessible from anywhere after declaration

```cpp
#include <iostream>

int g_var = 42;  // global variable (global scope)

void printGlobal() {
    std::cout << g_var << "\n"; // accessible anywhere
}

int main() {
    printGlobal();
}
```

Types of Scopes

- Global scope
- Namespace scope
  - visible only when qualified by that namespace or brought in with using



```cpp
#include <iostream>

namespace Math {
    int value = 10;  // namespace scope
    void print() {
        std::cout << "Value = " << value << "\n";
    }
}

int main() {
    Math::print();          // qualified access
    using namespace Math; // or bring names into current scope
    print();              // now accessible directly
}
```

Types of Scopes

- Global scope
- Namespace scope
- Block scope

```cpp
#include <iostream>

int main() {
    int x = 10;             // outer block scope
    {
        int y = 20;         // inner block scope
        std::cout << x + y << "\n"; // OK
    }
    std::cout << y;     // Error: y not in scope
}
```

Types of Scopes

- Global scope
- Namespace scope
- Block scope

```cpp
#include <iostream>

void greet(std::string name) { // parameter 'name' has function parameter scope
    std::cout << "Hello, " << name << "!\n";
}

int main() {
    greet("Denis");
    std::cout << name; //  Error: 'name' not visible here
}
```

- Function parameter scope

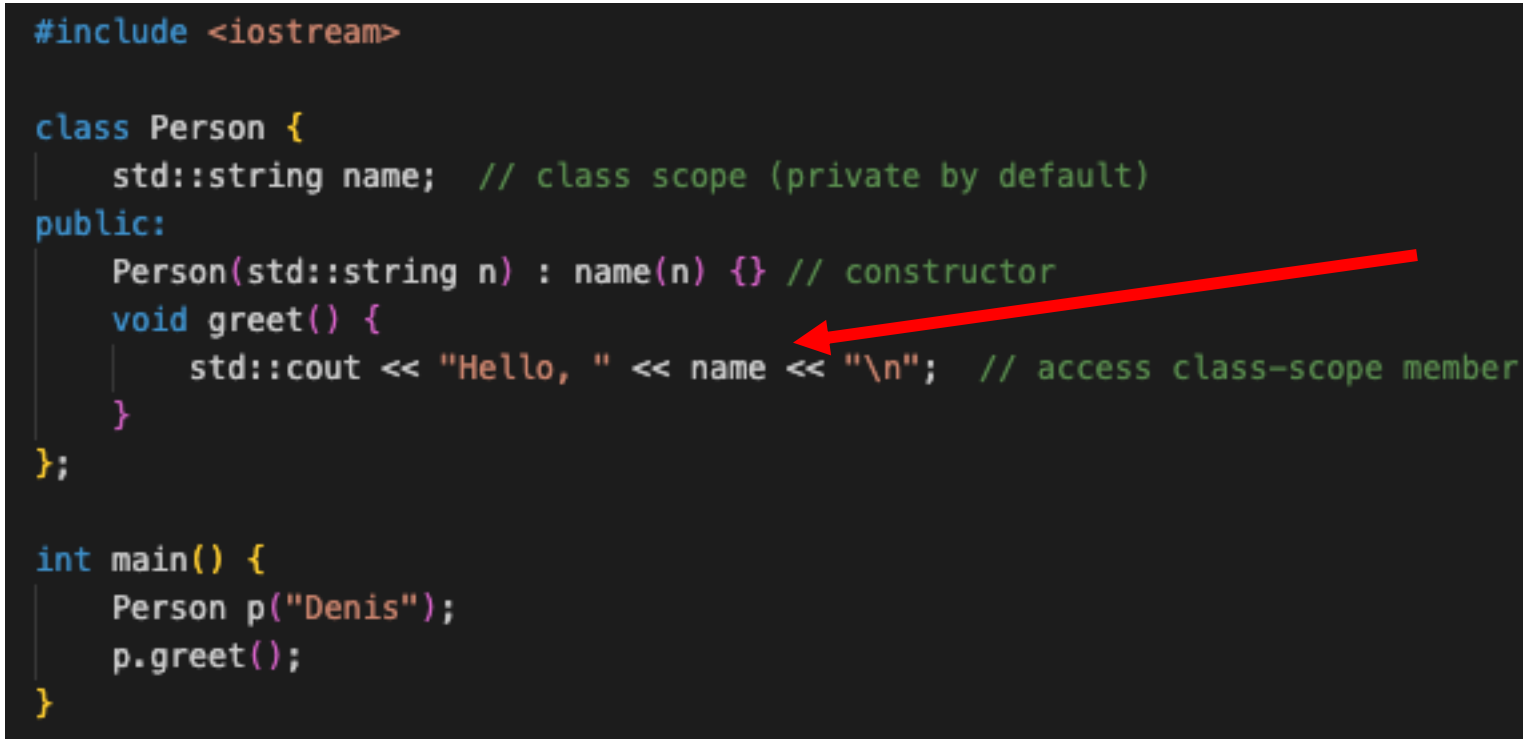
Types of Scopes

- Global scope
- Namespace scope
- Block scope
- Function parameter scope
- Statement scope
  - Example: if statement
- Class scope

```cpp
#include <iostream>

int main() {
    if (int n = 5; n > 3) {  // 'n' declared in statement scope
        std::cout << "n = " << n << "\n";
    }
    std::cout << n;       // Error: 'n' not visible here
}
```

Types of Scopes

- Global scope
- Namespace scope
- Block scope
- Function parameter scope
- Statement scope
- Class scope



```cpp
#include <iostream>

class Person {
    std::string name;  // class scope (private by default)
public:
    Person(std::string n) : name(n) {} // constructor
    void greet() {
        std::cout << "Hello, " << name << "\n";  // access class-scope member
    }
};

int main() {
    Person p("Denis");
    p.greet();
}
```

Example Static and class scope

- Objects declared static persist for the entire program lifetime but are visible only in their declared scope

```cpp
#include <iostream>

class Alpha {
public:
    static int value;  // static member, scoped to Alpha
};

class Beta {
public:
    static int value;  // static member, scoped to Beta
};

// Define both (required for static data members)
int Alpha::value = 10;
int Beta::value  = 20;

int main() {
    std::cout << "Alpha::value = " << Alpha::value << '\n';
    std::cout << "Beta::value  = " << Beta::value  << '\n';

    // Each static variable belongs to its class scope
    Alpha::value += 5;
    Beta::value  += 100;

    std::cout << "After modification:\n";
    std::cout << "Alpha::value = " << Alpha::value << '\n';
    std::cout << "Beta::value  = " << Beta::value  << '\n';
}
```

Instantiate

- How we have generated objects so far

```
1  struct Time
2  {
3    int hours_;
4    int minutes_;
5
6    ~Time() { std::cout << std::format("Delete time {}:{}\n", hours_, minutes_); }
7  };
8
9  int main()
10 {
11     Time time{17, 55};
12     return 0;
13 }
```

Instantiate

- How we have generated objects so far
- *Object is placed on the stack*

```
1  struct Time
2  {
3    int hours_;
4    int minutes_;
5
6    ~Time() { std::cout << std::format("Delete time {}:{}\n", hours_, minutes_); }
7  };
8
9  int main()
10 {
11     Time time{17, 55};
12     return 0;
13 }
```

Instantiate

- How we have generated objects so far
- *Object is placed on the heap*

Dynamic memory management

- new returns a pointer to the newly created object

```
1  struct Time
2  {
3    int hours_;
4    int minutes_;
5
6    ~Time() { std::cout << std::format("Delete time {}:{}\n", hours_, minutes_); }
7  };
8
9  int main()
10 {
11     Time* time = new Time{17,56};
12     return 0;
13 }
```

Lifetime

- Object on stack: lifetime ends automatically when scope ends
- Object on heap: lifetime ends manually

```cpp
struct Time
{
  int hours_;
  int minutes_;

  ~Time() { std::cout << std::format("Delete time {}:{}\n", hours_, minutes_); }
};

int main()
{
    Time* time = new Time{17,56};
    return 0;
}
```

Dynamic memory management

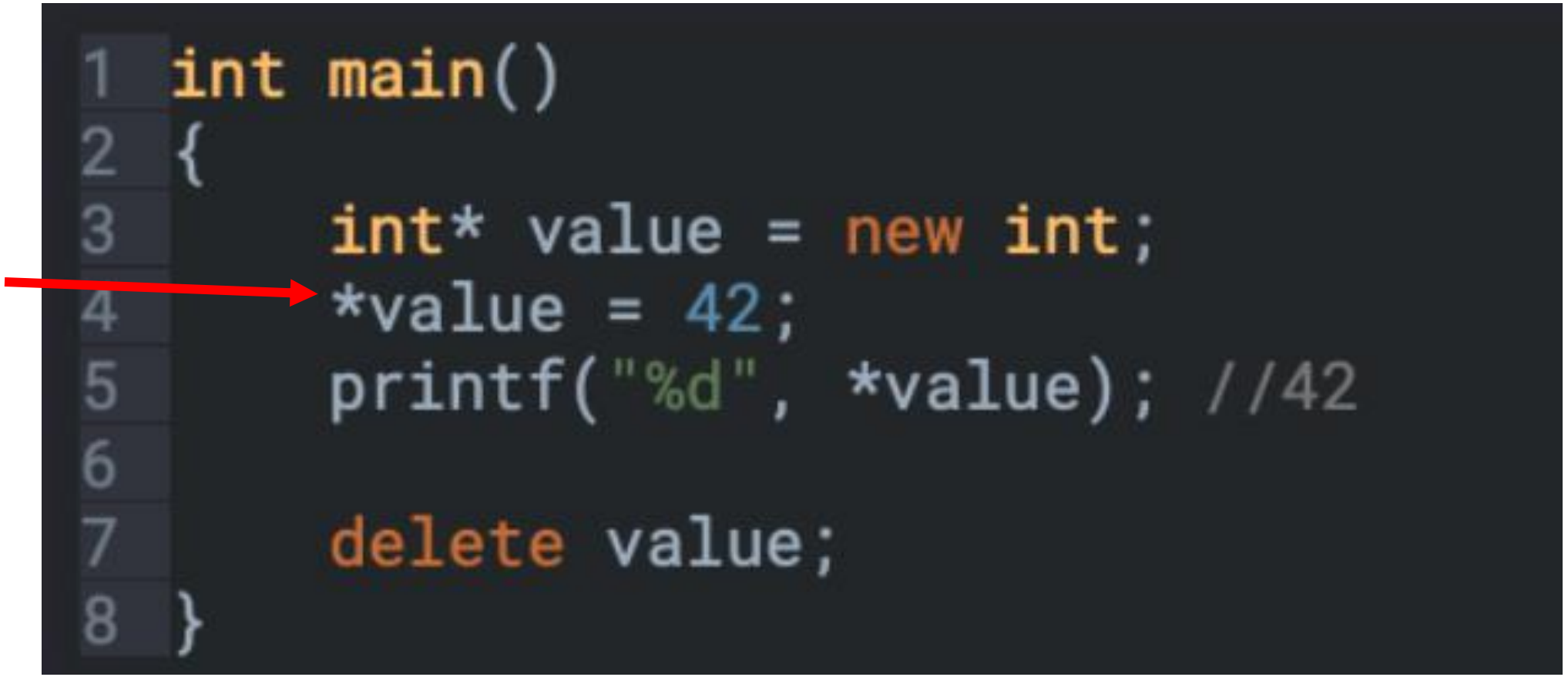
- new returns a pointer to the newly created object
- delete frees the memory again (and calls the destructor)

```
1  struct Time
2  {
3    int hours_;
4    int minutes_;
5
6    ~Time() { std::cout << std::format("Delete time {}:{}\n", hours_, minutes_); }
7  };
8
9  int main()
10 {
11     Time time = new Time{17,56};
12     delete time;
13     return 0;
14 }
```

Arrays of Objects

- Need delete[] so that every object is destroyed (destructor)
- delete (without []) on arrays does not call the destructors!

Access



```
1  int main()
2  {
3      int* value = new int;
4      *value = 42;
5      printf("%d", *value); //42
6
7      delete value;
8  }
```