

Object-Oriented Programming

References & Types II

Recap – OOP Concepts


- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Recap – Encapsulation

- Encapsulation provides **controlled access** and **protects data**

Recap – Encapsulation

- With access specifiers **public** / **private**
- Access to private members and methods is only possible over public interface

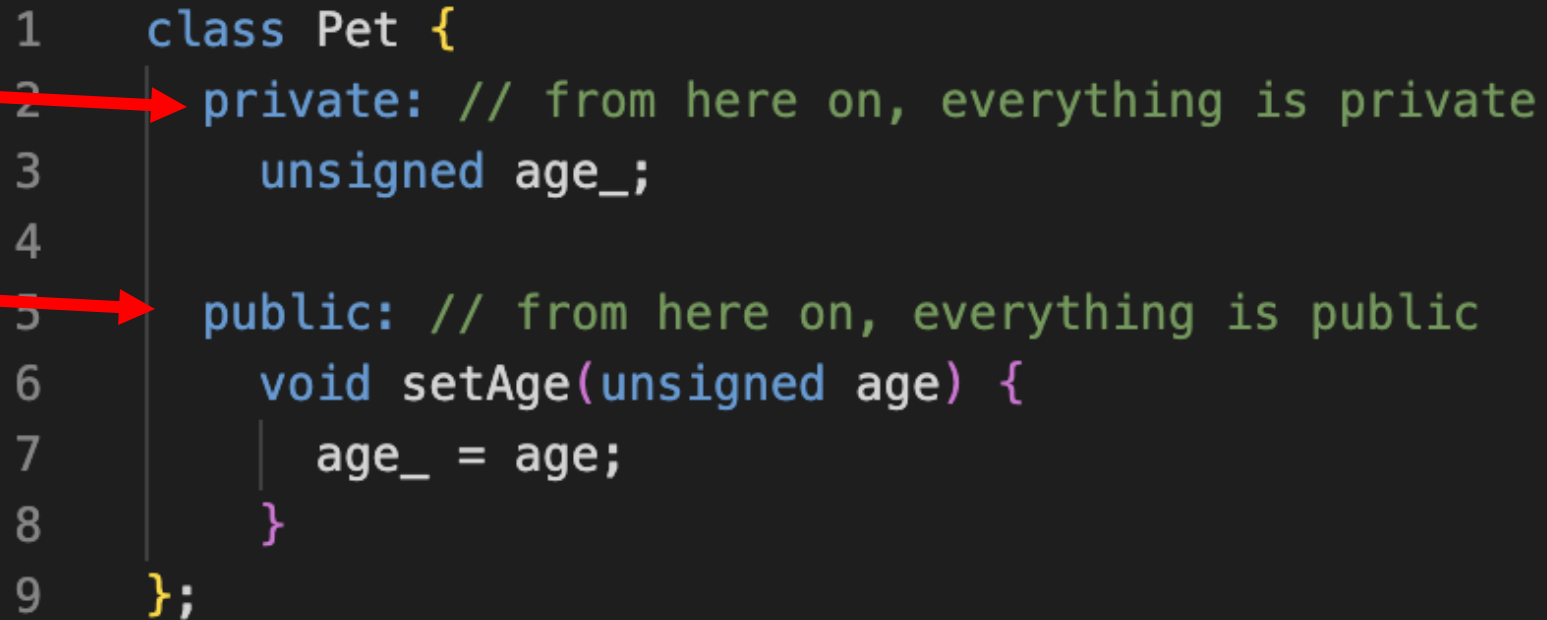


```
1  class Pet {  
2      private: // from here on, everything is private  
3          unsigned age_;  
4  
5      public: // from here on, everything is public  
6          void setAge(unsigned age) {  
7              age_ = age;  
8          }  
9  };
```

The image shows a C++ code snippet for a `Pet` class. The code is displayed on a dark background with syntax highlighting. Two red arrows point to the `private:` and `public:` access specifiers. The `private:` section (lines 2-3) contains a private member variable `age_`. The `public:` section (lines 5-8) contains a public method `setAge` that takes an `unsigned age` parameter and assigns it to `age_`. The class definition ends with a semicolon on line 9.

Recap – Encapsulation

- Prevents read and write access from outside the class
- Member functions can access all members of their own class

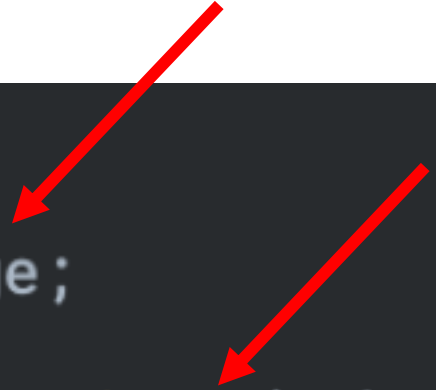


```
1  class Pet {  
2      private: // from here on, everything is private  
3          unsigned age_;  
4  
5      public: // from here on, everything is public  
6          void setAge(unsigned age) {  
7              age_ = age;  
8          }  
9  };
```

The image shows a C++ code snippet for a `Pet` class. The code is displayed on a dark background with syntax highlighting. Two red arrows point to the `private:` and `public:` access specifiers. The `private:` section contains a single member variable `unsigned age_;`. The `public:` section contains a member function `void setAge(unsigned age) { age_ = age; }`. The class definition is enclosed in curly braces and terminated with a semicolon.


Recap: PROBLEM - NAMING

```
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;}
      ~~~ ^ ~~~
```



Recap

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 slide. One arrow points to the variable name 'age_' on line 3, and the other points to the keyword 'unsigned' on line 5, highlighting the naming convention used in the code.

Recap

Solution 2: Initialization list

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



Recap 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 };
```



Recap

Memory Address

- Variables and functions are stored in memory
- Have an **address** in memory

```
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  }
```

Recap

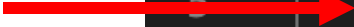
Memory Address

- Variables and functions are stored in memory

- Have an **address** in memory

- Address can be accessed using the address-of-operator **&**

```
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  }
```



Recap Pointer

```
1  #include <stdio>
2
3  int main() {
4      int number = 1;
5      int *ptr = &number;
6      printf("printf.1: %p\n", &number); //0x7fffffffddcd
7      printf("printf.2: %p\n", ptr); //0x7fffffffddcd
8      return 0;
9  }
```

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

Special Pointer – void*

- Generic pointer type that can hold the address of any data type

Example void*


- 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  }
```

Warning! void*

- If you pass the wrong type, you'll get **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  }
```



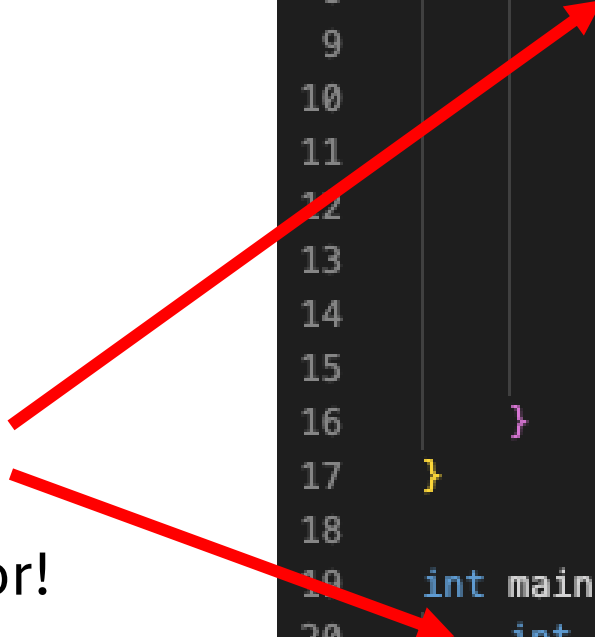
Warning! void*

- If you pass the wrong type, 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! void*

- 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 to Arrays

- 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 <csdio>
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 }
```



The diagram illustrates the concept of array decay. A red arrow points from the `College` struct definition (line 4) to the `College*` parameter in the `print_names` function (line 8). Another red arrow points from the `oxford` array variable in the `main` function (line 15) to the `print_names(oxford, ...)` call (line 16), demonstrating how the array name decays into a pointer to its first element.

Pointer for Array Decay

- When we use an array name, it is automatically converted to (or “decays into”) a pointer
- ‘Array-to-pointer decay’ refers to the automatic
 - conversion of an array
 - to a pointer to its first element.


```
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 'name' array in the 'College' struct (line 5) down to the 'colleges' parameter in the 'print_names' function (line 8), which is a pointer of type 'College*'. Another red arrow points from the 'oxford' array in the 'main' function (line 15) up to the 'print_names' function call (line 16), where 'oxford' is passed as an argument. This demonstrates how an array name decays into a pointer to its first element.

Pointer arithmetic

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

```
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

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

- Often used for looping through array

- 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 }
```



Warning! Pointer arithmetic

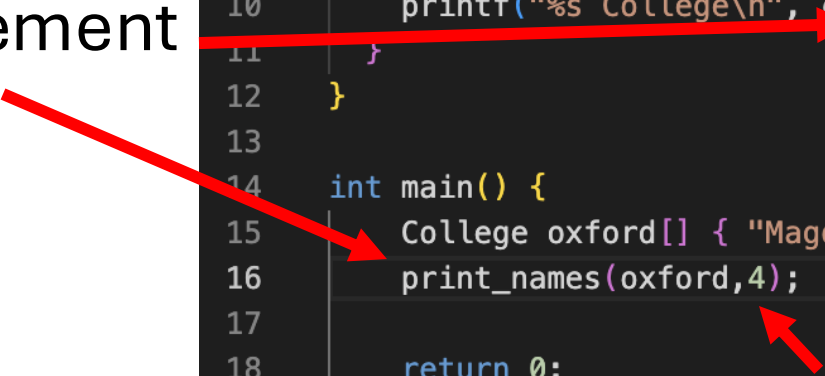
- Can be dangerous

```
1  #include <cstdint>
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 (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 }
```

Warning! Pointer arithmetic

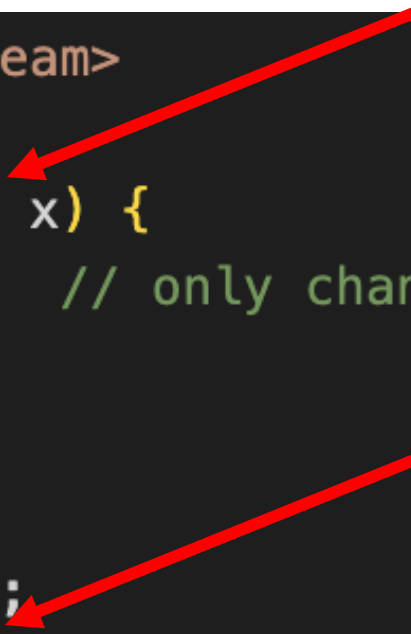
- Can be dangerous
- Need to make increment exists
- Random result or Prog. Crash

```
1  #include <cstdint>
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 (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 }
```

A diagram with two red arrows. One arrow starts from the text 'Need to make increment exists' and points to the 'college++' increment operation in the for loop of the print_names function. The second arrow starts from the text 'Random result or Prog. Crash' and points to the number '4' in the print_names function call in the main function, indicating that the loop will iterate 4 times over an array of 3 elements, causing a crash.

Function calls

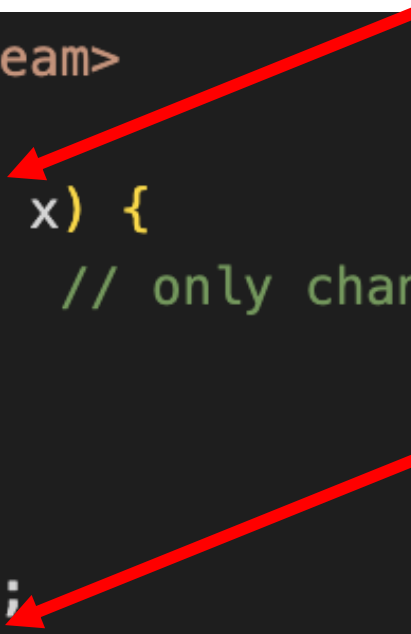
```
1  #include <iostream>
2
3  void addTen(int x) {
4      x = x + 10;  // only changes the copy
5  }
6
7  int main() {
8      int num = 5;
9      addTen(num);    // pass by value
10     std::cout << num << std::endl; // prints 5, not 15
11 }
```



Call-by-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 }
```



Function Call using Pointer

- call-by-value of a pointer



```
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

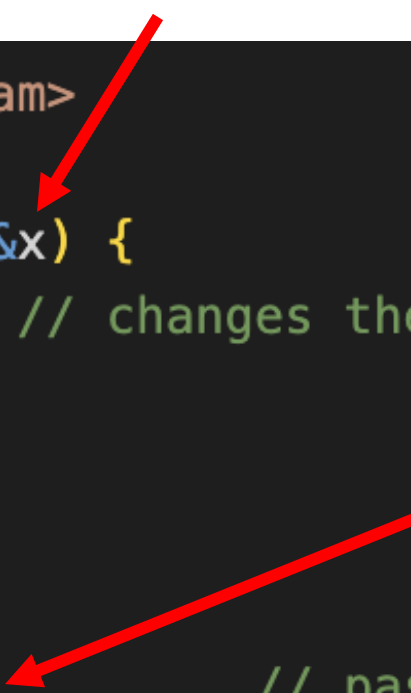
- Need to de-reference the address to modify variable

```
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 }
```

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 }
```

Two red arrows illustrate the call-by-reference mechanism. The first arrow originates from the parameter `&x` in the `addTen` function signature (line 3) and points to the variable `num` in the `addTen(num);` call (line 9). The second arrow originates from the `num` variable in the `main` function (line 8) and points to the `num` variable in the `std::cout` statement (line 10), indicating that the same memory location is accessed throughout.

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);
```

A red arrow originates from the right side of the image and points diagonally down and to the left, ending at the line of code '10 a_ref = b; // assign b to 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
```

Recap – Object Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Recap – Object Oriented Programming

- Encapsulation
- Abstraction
 - Expose essential behavior while hiding unnecessary details
- Inheritance
- Polymorphism

Abstraction

- Separate:
 - **Interface** (what operations are available)
 - **Implementation** (how those operations work)

Code Organization

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

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
```

Circular 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

#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
- **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
```

Forward Declaration

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

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

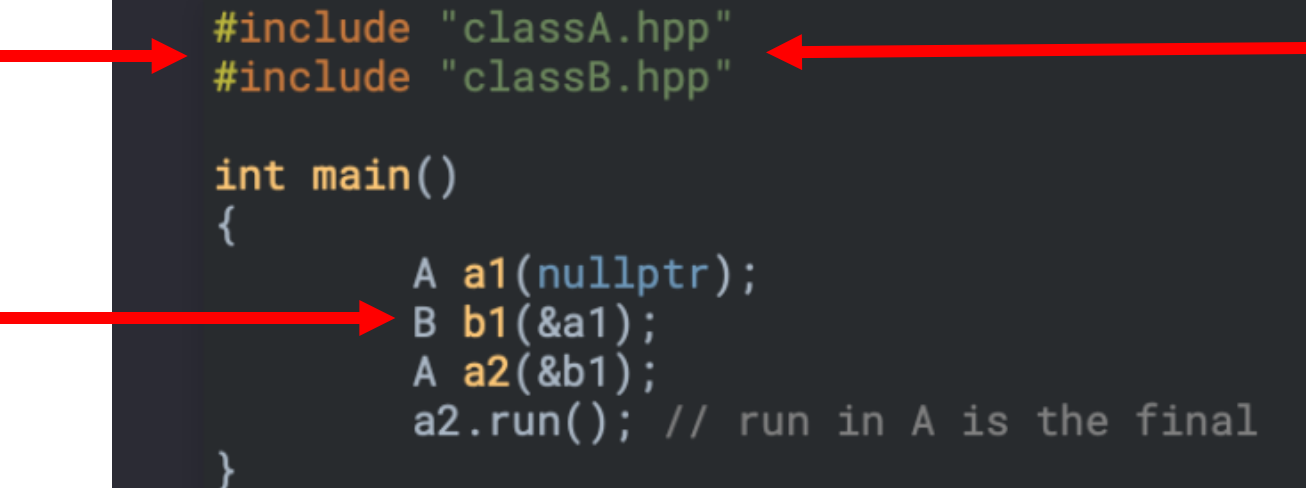
```
#include "classA.hpp"
#include "classB.hpp"
// now we know how both A and B look like!
void A::run()
{
    if(oldB_) oldB_>run();
    else printf("run in A is the final\n");
}
```

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
}
```



The diagram illustrates the include dependencies between the source files. Red arrows point from the `#include "classA.hpp"` line in `main()` to the `#include "classA.hpp"` line in `classA.cpp`. Another red arrow points from the `#include "classB.hpp"` line in `main()` to the `#include "classB.hpp"` line in `classB.cpp`. A third red arrow points from the `B b1(&a1);` line in `main()` to the `#include "classB.hpp"` line in `main()`, indicating that `classB.hpp` is needed to create the object `b1`.

static

- Static elements only exist **once** (not per object).
- Identified by the keyword static.

```
time.hpp

class Time
{
    /*...*/
public:
    static int minutesPerHour;
    static void timeFormat()
    {
        printf("hh:mm with %d minutes\n",
            minutesPerHour);
    }
    void displayTime();
    Time& setTime(double hours);
};
```

Two red arrows point to the lines `static int minutesPerHour;` and `static void timeFormat()` in the code block, highlighting the use of the `static` keyword for class-level variables and functions.

static


- Static elements only exist once (not per instance/object).
- They are identified by the keyword static.
- Access with scope operator :: when outside class header
- Example: **className::variableName**

static

time.hpp

```
class Time
{
    /*...*/
public:
    static int minutesPerHour;
    static void timeFormat()
    {
        printf("hh:mm with %d minutes\n",
            minutesPerHour);
    }
    void displayTime();
    Time& setTime(double hours);
};
```

time.cpp




```
#include "time.hpp"

int Time::minutesPerHour = 60;
void Time::displayTime()
{
    printf("It is %02d:%02d!\n",
        hour_, minute_);
}
Time& Time::setTime(double hours)
{
    hour_ = hours;
    minute_ = Time::minutesPerHour *
        (hours - hour_);
    return *this;
}
```

Example static

```
class Time
{
    /*...*/
public:
    static int minutesPerHour;
    static void timeFormat()
```

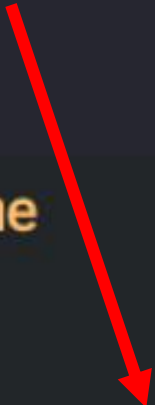
```
#include "time.hpp"
int main()
{
    Time a, b;
    Time::timeFormat(); //Access with scope operator > hh:mm with 60 minutes
    a.setTime(13.5).displayTime(); // It is 13:30!
    Time::minutesPerHour = 30;
    b.setTime(13.5).displayTime(); // It is 13:15!
}
```



Static Problem

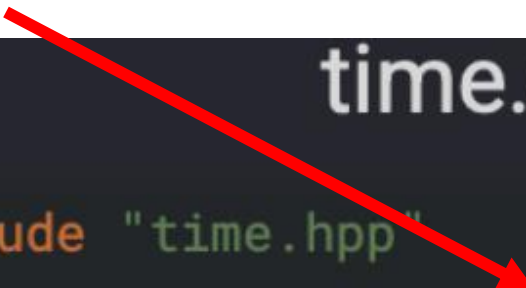
Definition not allowed inside class header

time.hpp



```
class Time
{
    /*...*/
public:
    static int minutesPerHour;
    static void timeFormat()
    {
        printf("hh:mm with %d minutes\n",
            minutesPerHour);
    }
    void displayTime();
    Time& setTime(double hours);
};
```

time.cpp



```
#include "time.hpp"

int Time::minutesPerHour = 60;
void Time::displayTime()
{
    printf("It is %02d:%02d!\n",
        hour_, minute_);
}
Time& Time::setTime(double hours)
{
    hour_ = hours;
    minute_ = Time::minutesPerHour *
        (hours - hour_);
    return *this;
}
```

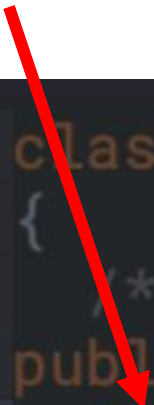
static - Problems

- Maintenance: declaration and definition scattered over separate files

static - Problems

- Maintenance: declaration and definition scattered over separate files
- Does not allow lightweight header-only classes

Since C++17 - inline static

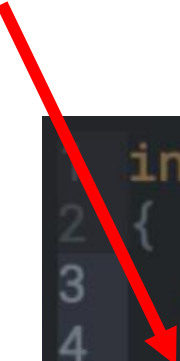


```
1 class Time
2 {
3     /*...*/
4 public:
5     inline static int minutesPerHour = 60;
6     static void timeFormat()
7     {
8         printf("hh:mm mit %d Minuten\n",
9             minutesPerHour);
10    }
11    void displayTime();
12    void setTime(double hours);
13};
```

- Declared and defined in the header

const

- Compiler prohibits modifications



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

const

- Constants must be initialized upon definition

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


const

- Constants must be initialized upon definition

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

→ **ERROR**

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

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 !!!

const-Attributes

- Constants must be initialized upon definition
 - In header

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

const-Attributes

- Constants must be initialized upon definition
 - In header

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

Forbidden!

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

const-Attributes

- Constants must be initialized upon definition

- In header

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

- Using initializer list

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

const or #define?

`const unsigned age{ 20 };`

better than:

`#define age 20`

const or #define?

const unsigned age{ 20 };

better than:

#define age 20

- Reasons:
 - const allows the compiler to perform type checks

const or #define?

`const unsigned age{ 20 };`

better than:

`#define age 20`

- Reasons:
 - const allows the compiler to perform type checks
 - debugging is easier with const

const or #define?

`const unsigned age{ 20 };`

better than:

`#define age 20`

- Reasons:
 - const allows the compiler to perform type checks
 - debugging is easier with const
 - If two headers both define `#define TESTVAR 3`, whoever is included last wins => silently

const or #define?

`const unsigned age{ 20 };`

better than:

`#define age 20`

- Reasons:
 - const allows the compiler to perform type checks
 - debugging is easier with const
 - #define defines a symbol for the entire file
 - watch out when you #include a file with a define...

```
#define VALUE 10
#include "other.hpp"
int x = VALUE;
```

VALUE is 10 **inside other.hpp**
because it was included *after* the define

const or #define?

`const unsigned age{ 20 };`

better than:

`#define age 20`

- Reasons:
 - const allows the compiler to perform type checks
 - debugging is easier with const
 - #define defines a symbol for the entire file
 - watch out when you #include a file with a define...

```
#define VALUE 10
#include "other.hpp"
int x = VALUE;
```

VALUE is 10 inside other.hpp,

int VALUE = 5 // becomes int 10 = 5