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

```
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};
```

Recap – Encapsulation

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

```cpp
class Pet {
  private: // from here on, everything is private
    unsigned age_;

  public: // from here on, everything is public
    void setAge(unsigned age) {
      age_ = age;
    }
};
```

Recap: PROBLEM - NAMING

```
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) {age = age;}
};
```

Compiler output

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

Recap

Better: use naming convention

```
class Cat
{
  unsigned age_;
  public:
    Cat(unsigned age) { age_ = age; }
};
```

Recap

Solution 2: Initialization list

```cpp
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) : age{age} {}
};
```

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)

```cpp
class Cat
{
  unsigned age;
  public:
    Cat(unsigned age) { this->age = age; }
};
```

Recap
Memory Address

- Variables and functions are stored in memory

- Have an **address** in memory

```
#include <cstdio>

int main() {
    int number = 1;
    int *ptr = &number;
    printf("printf.1: %p\n", &number); //0x7fffffffdcdc
    printf("printf.2: %p\n", ptr); //0x7fffffffdcdc
    return 0;
}
```

Recap
Memory Address

```cpp
#include <cstdio>

int main() {
    int number = 1;
    int *ptr = &number;
    printf("printf.1: %p\n", &number); //0x7fffffffdcdc
    printf("printf.2: %p\n", ptr); //0x7fffffffdcdc
    return 0;
}
```

- Variables and functions are stored in memory
- Have an **address** in memory
- Address can be accessed using the address-of-operator **&**

Recap
Pointer

```
#include <cstdio>

int main() {
    int number = 1;
    int *ptr = &number;
    printf("printf.1: %p\n", &number); //0x7ffffffdcdc
    printf("printf.2: %p\n", ptr); //0x7ffffffdcdc
    return 0;
}
```

- 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

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Warning! void*

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

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Warning! void*

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

- For example:

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

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

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

```cpp
#include <iostream>

void printValue(void* ptr, char type) {
    switch (type) {
        case 'i': // int
            std::cout << *(int*)ptr << std::endl;
            break;
        case 'd': // double
            std::cout << *(double*)ptr << std::endl;
            break;
        case 'c': // char
            std::cout << *(char*)ptr << std::endl;
            break;
        default:
            std::cout << "Unknown type!\n";
    }
}

int main() {
    int a = 42;
    double b = 3.14;
    char c = 'Z';

    printValue(&a, 'i');
    printValue(&b, 'd');
    printValue(&c, 'c');
}
```

Pointer to Arrays

- When we use an array name, it is automatically converted to (or “decays into”) a pointer

- College* colleges = &oxford[0];

```cpp
#include <cstddef>
#include <cstdio>

struct College {
    char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
    for (size_t i = 0; i < n_colleges; i++) {
        printf("%s College\n", colleges[i].name);
    }
}

int main() {
    College oxford[] { "Magdalen" , "Nuffield" , "Kellogg" };
    print_names(oxford, sizeof(oxford) / sizeof(College));
}
```

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.

```
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (size_t i = 0; i < n_colleges; i++) {
    printf("%s College\n", colleges[i].name);
  }
}

int main() {
    College oxford[] { "Magdalen" , "Nuffield" , "Kellogg" };
    print_names(oxford, sizeof(oxford) / sizeof(College));
}
```

Pointer arithmetic

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

```cpp
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,3);

    return 0;
}
```

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)

```cpp
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,3);

    return 0;
}
```

Warning! Pointer arithmetic

- Can be dangerous

```
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,4);

    return 0;
}
```

Warning! Pointer arithmetic

- Can be dangerous

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

```cpp
#include <cstddef>
#include <cstdio>

struct College {
  char name[256];
};

void print_names(College* colleges, size_t n_colleges) {
  for (College* college = colleges; college < colleges + n_colleges; college++) {
    printf("%s College\n", college->name);
  }
}

int main() {
    College oxford[] { "Magdalen", "Nuffield", "Kellogg" };
    print_names(oxford,4);

    return 0;
}
```

Function calls

```cpp
#include <iostream>

void addTen(int x) {
    x = x + 10;  // only changes the copy
}

int main() {
    int num = 5;
    addTen(num);           // pass by value
    std::cout << num << std::endl; // prints 5, not 15
}
```

Call-by-Value

- A copy is passed

```cpp
#include <iostream>

void addTen(int x) {
    x = x + 10;  // only changes the copy
}

int main() {
    int num = 5;
    addTen(num);            // pass by value
    std::cout << num << std::endl; // prints 5, not 15
}
```

Function Call using Pointer

- call-by-value of a pointer

```cpp
#include <iostream>

void addTen(int *x) {
    *x = *x + 10;  // dereference pointer to change original
}

int main() {
    int num = 5;
    addTen(&num);   // pass address
    std::cout << num;     // prints 15
}
```

Call-by-Value using Variable Address

- Need to de-reference the address to modify variable

```cpp
#include <iostream>

void addTen(int *x) {
    *x = *x + 10;  // dereference pointer to change original
}

int main() {
    int num = 5;
    addTen(&num);   // pass address
    std::cout << num;     // prints 15
}
```

Call-by-Reference

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

```cpp
#include <iostream>

void addTen(int &x) {
    x = x + 10;  // changes the original variable
}

int main() {
    int num = 5;
    addTen(num);          // pass by reference
    std::cout << num << std::endl; // prints 15
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a);
    printf("%d\n", a_ref);
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref);
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref); //24601
    printf("%p\n", a_ptr);
}
```

Examples – What is the output?

```
#include <cstdio>

void main () {
    int a = 24601;
    int& a_ref = a;
    int* a_ptr = &a;

    printf("%d\n", a); //24601
    printf("%d\n", a_ref); //24601
    printf("%p\n", a_ptr); //0x7ffffffdcdc
}
```

Examples – What is the output?

```cpp
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – What is the output?

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – References & Pointers

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref);
printf("%d\n", a);
printf("%p\n", &b);
printf("%p\n", &a_ref);
```

Examples – References & Pointers

```
int a = 24601;
int b = 24602;
int& a_ref = a;
int* a_ptr = &a;

printf("%d\n", a); //24601
printf("%d\n", a_ref); //24601
printf("%p\n", a_ptr); //0x7fffffffdcc8

a_ref = b; // assign b to a_ref

printf("%d\n", a_ref); //24602 <- value of the reference has changed
printf("%d\n", a); //24602 <- value of a has changed
printf("%p\n", &b); //0x7fffffffdcc4
printf("%p\n", &a_ref); //0x7fffffffdcc8 <- 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

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

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

Time.cpp

```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
}
```

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);
};
```

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

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

```cpp
#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

```cpp
class Time
{
  /*...*/
public:
  inline static int minutesPerHour = 60;
  static void timeFormat()
  {
        printf("hh:mm mit %d Minuten\n",
          minutesPerHour);
  }
  void displayTime();
  void setTime(double hours);
};
```

- Declared and defined in the header

const

- Compiler prohibits modifications

```
int main()
{
  int babe_age{ 0 };
  babe_age++; // OK
  const int dorian_gray_age{ babe_age + 19 }; // initialize constant to 20
  dorian_gray_age++; // error: cannot assign to variable
                     //        with const-qualified type 'const int'
  return 0;
}
```

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