Contents

1. Static memory
2. Object oriented programming I
Memory classes

- Stack memory
  - Local variables / automatic variables

- Heap memory
  - Variables allocated with `new`

- Static memory
  - Static because its size does not change at runtime!
  - Stored in programs data segment
  - Global variables are static
  - Variables in functions, classes / structs and modules can be declared static
    - A static variable is a global variable that cannot be accessed globally!
Using static variables in classes

```cpp
struct S {
    int i;
    static const int j = 42;
    S(int a) : i(a) {} // Non constant static variable are not allowed
    friend ostream& operator<< (ostream& os, const S& s) {
        return os << s.i << " " << s.j; }
};

int main() {
    S s(10);
    cout << s << "\n";
    S t(20);
    cout << t << "\n";
    cout << s.j << "\n";
    cout << t.j << "\n";
    cout << S::j << "\n";
    return 0;
}
```

- Classes / structs can contain constant static variables
- Non constant static variable are not allowed
- j is shared across all variables of type S
- Does not require an instance to access it
- Why is that useful?
  - Constant variables are only stored once
  - Saves memory
  - (Template metaprogramming)
Static variables in functions

#include <iostream>
using namespace std;

void function() {
    // counter is initialized only once
    static size_t counter = 0;
    counter++;
    cout << "called " << counter
         << " times!\n";
}

int main() {
    size_t n = 10;
    while (n--)
        function();
    return 0;
}
Static variables and static (non-member-)functions in modules

// module.h
#ifndef SOME_HEADER_H_
#define SOME_HEADER_H_

static int value;
static void foo();

#endif

// module.cpp
int value = 42;

void foo() {}
Special member: \texttt{this} pointer

- **Pointer**
- Contains address of object on which the member function is being called on
- Where can the this pointer be used?
  - Within non-static member functions
  - … (and two more)

```cpp
class T {
    int x, y;
    void foo() {
        x = 6;    // same as this->x = 6;
        this->x = 5;    // explicit use of this->
    }
    // parameter x shadows the member with the same name
    void foo(int x) {
        this->x = x;    // unqualified x refers to the parameter
                        // 'this->' used for disambiguation
    }
    T(int x) : x(x), // uses parameter x to initialize member x
               y(this->x) // uses member x to initialize member y
    {}
    T& operator= ( const T& b ) {
        x = b.x;
        return *this;    // many overloaded operators return *this
    }
};
```

Ideas of OO-Languages

- “Concept of describing ‘real world’ objects.”
  - Kind of (nonsense), and the least important idea!

- Three foundations
  1. Encapsulation
  2. Inheritance
  3. Polymorphism
Spoiler

- Foundations of OOP
  1. Encapsulation
  2. Inheritance
  3. Polymorphism

- You have to understand the concepts and the ideas!
  - But you will only need the above when designing & implementing large software systems
  - Almost never happens in University
  - Be prepared for reality
  - Know the mechanisms
Model objects using classes and structs

- What is an object in C++?
  - “An object is a region of storage.”

- Object oriented thinking
  - For a real-world object
    - Model it as a class / struct
    - Abstract away from a concrete object and model essential properties as data members
    - Model capabilities of an object using member functions
    - An object might interact with other objects
  - Example: “not so real-world object” matrix from mathematics
    - Has a number of rows, columns, some entries
    - Can be multiplied with another matrix object
Model objects using classes and structs

- Another example
  - A triangle from $\mathbb{R}^3$
    - Properties
      - Three points
      - ...
    - Capabilities
      - Calculate area
      - Calculate circumference
      - ...
    - Interaction
      - Calculate intersections
      - ...
  - ... as it is used in computer graphics

- Model an object with a class / struct
  - Description
  - Blueprint

- Creating a variable of that class / struct
  - Creates an instance of that blueprint
  - Obtain a concrete object
Encapsulation

- “Hide” programming details
  1. Separation of data & interface using …
     - public
     - private
     - protected
  2. Separation of interface & implementation using header / implementations files

- Example
  - TVs
Encapsulation

- **Header file matrix.h**

```cpp
#ifndef MATRIX_H
#define MATRIX_H
#include <iostream>
#include <vector>
using namespace std;

class matrix {
private:
    size_t rows;
    size_t columns;
    vector<double> data;
public:
    matrix(size_t rows, size_t cols);
    size_t get_rows();
    size_t get_columns();
    size_t get_elements();
    friend matrix operator* (const matrix& lhs, const double scale);
    friend matrix operator* (const matrix& lhs, const matrix& rhs);
};
#endif
```

- **Implementation file matrix.cpp**

```cpp
#include "matrix.h"
matrix::matrix(size_t rows, size_t cols) :
    rows(rows), columns(cols), ...
size_t matrix::get_rows() {
    // ...
}
size_t matrix::get_columns() {
    // ...
} // other function implementations
```
Encapsulation

- User only need to know the interface
- Concrete implementation is “hidden”
- User does not recognize a change in the implementation
- Implementation can easily be changed

main.cpp
uses matrix type

matrix.h

matrix.cpp

matrix.o

better_matrix.cpp
Better implementation

main.o

better_matrix.o
Inheritance

- Classes / structs can inherit from each other
- Results in inheritance of features
- Child classes inherit capabilities of parents
  - Child obtains
    - Data members
    - Function members
  - Child classes / structs can be further extended and / or specialized
- Inheritance follows an “is a” structure
  - A duck is some kind of bird
    - A bird is some kind of animal
      - An animal is some kind of creature
Inheritance

- Inheritance of data members & function members
  - Saves typing same stuff over and over again
  - About reusability
    - Imagine a change in a class has to be made
      - Extension to class logic
    - Copy version:
      - Terrible
    - Inheritance version:
      - Much easier
Inheritance

```cpp
#include <iostream>
using namespace std;

class parent {
private:
  int parent_value;

public:
  parent(int i) : parent_value(i) {}
  void print() {
    cout << parent_value << 'n';
  }
};

int main() {
  parent p(10);
  p.print();

  child c (20, 30);
  c.print();
  return 0;
}
```
Inheritance

- Visibility / inheritance modes
  - `public`
  - `private`
  - `protected`

- Variables of base
  - Everything aware of base is aware of `i`
  - Only children of base (and their children) are aware of `k`
  - No one, but base is aware of `j`

- Inheritance modes
  ```cpp
  class derived1 : public base {
    // everything is aware that derived1 inherits from base
  };
  class derived2 : private base {
    // no one but derived2 is aware of that inheritance
  };
  class derived3 : protected base {
    // only child and its children are aware that they inherit from base
  };
  ```

- Most of the time nobody cares and `public` inheritance mode is used
Build a class hierarchy

Shape

Triangle

Circle

Rectangle

Right Triangle
Build a class hierarchy

class shape {};

class triangle : public shape {};

class rectangle : public shape {};

class circle : public shape {};

class right_triangle : public triangle {};
Build a class hierarchy

```cpp
class shape {
public:
    void draw() { cout << "shape\n"; }
};
class circle : public shape {};
class rectangle : public shape {};
class triangle : public shape {};
class right_triangle : public triangle {};
```

- It would be nice if we can draw the shapes!
- Provide a draw function
- Be smart
  - Make use of inheritance!
  - Now every child can call `draw()`
- But wait
  - Not all shapes shall be printed as a shape
  - We want to specialize `draw()`
    - Depending on the subtype
Build a class hierarchy

```cpp
class shape {
public:
    virtual ~shape() = default;
    virtual void draw() { cout << "shape\n"; }
};
class circle : public shape {
public:
    void draw() override { cout << "circle\n"; }
};
class rectangle : public shape {
public:
    void draw() override { cout << "rectangle\n"; }
};
class triangle : public shape {
public:
    virtual void draw() override { cout << "triangle\n"; }
};
class right_triangle : public triangle {
public:
    void draw() override { cout << "R triangle\n"; }
};
```

- We want to specialize `draw()`
- Depending on the subtype
- Make `draw` virtual
- `virtual` functions can be overridden!
- That is called specialization!
- Be smart
- Use `override` to denote specialized functions
- Not using `override` leads to danger
Polymorphism

- Why is it useful to specialize functions (override them)?
- Doesn`t this result in?
  
  No!
  
  Because
  
  - We have specialization
  - We do not have to override
  - Achieved polymorphism!
    
    - Types in that hierarchy behave polymorphic
    - Example class circle
      
      circle has type circle, but also type shape (since it inherits from shape)
Polymorphism

```cpp
#include <iostream>
using namespace std;

void print_shape(const shape& s) {
    s.draw();
}

int main() {
    circle c;
    right_triangle rt;
    print_shape(c);
    print_shape(rt);
    return 0;
}
```

```cpp
class shape {
    public:
        virtual ~shape() = default;
        virtual void draw() { cout << "shape\n"; }
    }

class circle : public shape {
    public:
        void draw() override { cout << "circle\n"; }
    }

class rectangle : public shape {
    public:
        void draw() override { cout << "rectangle\n"; }
    }

class triangle : public shape {
    public:
        virtual void draw() override { cout << "triangle\n"; }
    }

class right_triangle : public triangle {
    public:
        void draw() override { cout << "R triangle\n"; }
    }
```
Caution: need for virtual destructors

```cpp
class shape {
public:
    virtual ~shape() = default;
    virtual void draw() { cout << "shape\n"; }
};
```

- If a C++ type is meant to be used polymorphically its destructor has to be declared virtual. Otherwise, if the static type of an object to be deleted differs from its dynamic type, the behavior is undefined.

- Example
```cpp
int main() {
public:
    shape *s = new circle;
    s->draw();
    delete s;
    return 0;
};
```
Polymorphism

```cpp
#include <iostream>
using namespace std;

void print_shape(const shape& s) {
    s.draw();
}

void print_triangle(const triangle& t) {
    t.draw();
}

int main() {
    circle c;
    right_triangle rt;
    print_shape(c);
    print_triangle(rt);
    return 0;
}
```

- `print_shape()` accepts a shape and any of its subtypes
- `print_triangle()` accepts a triangle and any of its subtypes
- Note that we pass by reference!
- Such behavior is only possible when passing by
  A. Reference
  B. Pointer
- Why not by value?
  ```cpp
  void print_shape(shape s) {
      s.draw();
  }
  ```
  - Accepts a shape?
  - Yes but creates a copy of a shape which is type `shape`!
  - Pass any subtype results in passing a shape
  - Would print “shape” every time
Polymorphism

Example using smart pointers

```cpp
#include <iostream>
#include <memory>
#include <random>
using namespace std;

struct shape
{
    virtual ~shape() = default;
    virtual void draw() { cout << "shape\n"; }
};

struct rectangle : shape
{
    void draw() override { cout << "rectangle\n"; }
};

struct circle : shape
{
    void draw() override { cout << "circle\n"; }
};

void print_shape(shared_ptr<shape> s) { s->draw(); }

vector<shared_ptr<shape>> rand_shapes(size_t n) {
    vector<shared_ptr<shape>> shapes;
    mt19937 rng(1);
    uniform_int_distribution<int> gen(0, 2);
    while (n--) {
        switch (gen(rng)) {
            case 0: shapes.push_back(make_shared<shape>()); break;
            case 1: shapes.push_back(make_shared<rectangle>()); break;
            case 2: shapes.push_back(make_shared<circle>()); break;
        }
    }
    return shapes;
}

int main() {
    auto shapes = rand_shapes(10);
    for (auto shape : shapes)
        print_shape(shape);
    return 0;
}
```
Static polymorphism versus dynamic polymorphism

- Static binding (at compile time)

```cpp
struct shape {
    void draw() { cout << "shape\n"; }
};
struct rectangle {
    void draw() { cout << "rectangle\n"; }
};
void print_shape(const shape &s) {
    s.draw(); // binds statically
}
void print_shape(const rectangle &s) {
    s.draw(); // binds statically
}
int main() {
    shape s;
    rectangle r;
    print_shape(s);
    print_shape(r);
    return 0;
}
```

- Dynamic binding (at runtime)

```cpp
struct shape {
    virtual ~shape() = default;
    virtual void draw() { cout << "shape\n"; }
};
struct rectangle : shape {
    void draw() override { cout << "rectangle\n"; }
};
void print_shape(const shape &s) {
    s.draw(); // cannot bind statically
}
int main() {
    shape s;
    rectangle r;
    print_shape(s);
    print_shape(r);
    return 0;
}
```
Virtual function table / vtable

```cpp
#include <iostream>
using namespace std;
struct A {
    virtual ~A() = default;
    virtual void print() { cout << "A\n"; }
    virtual void other() { cout << "other\n"; }
};
struct B : A {};
struct C : A {
    void print() override { cout << "C\n"; }
};

// How can call_print() choose the right print() function?
// size and layout of B and C is UNKOWN to call_print()!
void call_print(A& a) {
    a.print();
}
```

- A has virtual functions
- Virtual function table / vtable / vtbl is used
- call_print() only needs to follow vtbl pointer
- Index used for each virtual function
- Efficiency compared to normal function call:
  - Time: within 25%, but in reality much faster
  - Space: one pointer to vtbl per variable + one vtbl per type
Virtual method table
(Idea)

```cpp
class Base {
public:
    virtual ~Base() = default;
    virtual void function1() {};
    virtual void function2() {};
};

class D1: public Base {
public:
    void function1() override {};
};

class D2: public Base {
public:
    void function2() override {};
};

class Base {
    public:
        FunctionPointer *__vptr;
        virtual void function1() {};
        virtual void function2() {};
    }

class D1: public Base {
    public:
        void function1() override {};
    }

class D2: public Base {
    public:
        void function2() override {};
    }
```

[Figure taken from http://www.learncpp.com/cpp-tutorial/125-the-virtual-table/]
Special case: Interfaces

- Abstract classes / structs
  - Miss one or more function implementation
  - Cannot be instantiated
  - An interface is an abstract class with missing function definitions!
- Why is there a need for interfaces?
- Example
  - A bird might quack
  - But it is not useful to inherit from base class “quacker”
  - Make it an interface!
    - A duck might inherit from bird
      - And additionally implements the quack interface
- An interface specifies what has to be implemented
Special case: Interfaces

```
#include <iostream>
using namespace std;

class bird {
public:
    virtual ~bird() = default;
    virtual void fly() { cout << "fly
"; }
};

class quack {
public:
    virtual ~quack() = default;
    //provide no implementation
    virtual void do_quack() = 0;
};

class duck : public bird, public quack {
public:
    void quack() override { cout << "quack, quack!
"; }
};
```

```c
int main() {
    duck d;
    d.fly();
    d.do_quack();
    // the following line does not work
    quack q; // cannot instantiate abstract class!
    return 0;
}
```
Multiple inheritance

- More than one parent is possible
- Quite useful
- Some languages do not support that, considered as dangerous (Java)
  - “Deadly diamond of death” (diamond problem)

```cpp
#include <iostream>
using namespace std;
struct A {
    virtual ~A() = default;
    virtual void print() { cout << "A\n"; }
};
struct B : A { void print() override { cout << "B\n"; } }
struct C : A { void print() override { cout << "C\n"; } }
struct D : B, C {};
int main() {
    D d;
    d.print();
    return 0;
}
```

- What `print()` should be called?
- C++ warns you when this happens
- Problem can be mitigated
  - use virtual inheritance or override `print()` in `D` and manually direct the call
Multiple inheritance

- “Deadly diamond of death” (diamond problem) → fixed

```cpp
#include <iostream>
using namespace std;
struct A {
    virtual ~A() = default;
    virtual void print() { cout << "A\n"; };
}
struct B : A { void print() override { cout << "B\n"; } ;
struct C : A { void print() override { cout << "C\n"; } ;
struct D : B, C {
    void print() override { B::print(); }
};
int main() {
    D d;
    d.print();
    return 0;
}
```
Nice talks about OOP

- “Intro to the C++ Object Model”, Richard Powell (CppCon 2015)
  - https://www.youtube.com/watch?v=iLiDezv_Frk

- A more general talk “SOLID Principles of Object Oriented & Agile Design”, Robert Martin (Yale school of management 2014)
  - https://www.youtube.com/watch?v=TMuno5RZNeE
Thank you for your attention

Questions?