Outline

Classes and Objects

Constructors and Destructors

References

Function, Method and Operator Overloading

Virtual Methods

Exceptions

Templates
Idea: an object is an *instance* of class

- John is an instance of a person

- Peter is also in instance of a person, but he is not John

We’ll call John and Peter objects, and say that “person” is a class.
Idea: objects have attributes

- John’s name is “John”. His gender is MALE. His age is 43.
- Peter’s name is “Peter”. His gender is MALE. His age is 13.

These attributes apply to all persons, and we don’t want to re-do the list for each object.
class Person {
    string name;
    enum { MALE, FEMALE } gender;
    unsigned age;
};

This is much like a struct in C, but we can omit the aggregate key when using the type.
Idea: we often want to specialise or extend a class with further information.

- All students are persons, but not all persons are students. There is a student number to be stored.

- There are also teachers, who are persons as well. We want to store a list of courses taught.
class Student: public Person {
    string student_number;
};

class Teacher: public Person {
    list<Course> courses_taught;
};
Type Hierarchies

Note that we can use any Student or Teacher as a Person.

Student John;
Teacher Mike;

Person someone1=John;
Person someone2=Mike;

The opposite does not work.
Member Functions

Code should be linked to the data it operates on.

Thus, don’t do this:

```cpp
class Person {
    string name;
    enum { MALE, FEMALE } gender;
    unsigned age;
};

void birthday(Person *p) {
    p->age += 1;
}
```
class Person {
    string name;
    enum { MALE, FEMALE } gender;
    unsigned age;

    void birthday();
};

void Person::birthday()
{
    age+=1;
}
Member Functions

- Member functions are implicitly given a pointer to the object
- The pointer is accessible with the keyword `this`
- The pointer can be made a pointer to a constant object by adding `const` after the method parameters

```cpp
class Person {
    string name;
    enum { MALE, FEMALE } gender;
    unsigned age;
    bool is_cool() const { return age<20; }
};
```
Access Protection

- Members of classes can be public, private or protected

- **public**: Anyone can access
- **private**: Only the member functions of the class can access
- **protected**: Only the member functions of the class or derived classes can access

- The default in a class is protected

- You can also declare other classes to be friends
Constructors

- Constructors contain “set up” or “initialization” code for a class

- They do not return a value, not even void

- Constructors can have parameters, and if there is exactly one, then the compiler will use the constructor implicitly

- If this is potentially confusing, add the keyword explicit
class Person {
    public:
    Person(string s) {
        name = s;
    }

    private:
    string name;
};
Constructors

Alternatively:

```cpp
class Person {
public:
    Person(string s): name(s) {
    }

private:
    string name;
};
```
Let's use it:

```java
int main()
{
    Person Mike("Mike");
    Person John("John");

    string name="Pete";
    Person someone=name;
}
```
Default Constructors

▶ If you don’t define a constructor, the compiler will make one for you.

▶ The default constructor will call the default constructors for all base classes and all member objects.

▶ It also creates a *copy constructor*, which takes one parameter that has the same type as the class. It copies all members.
Destructors

There is sometimes clean-up work to do.

class temporary_file {
public:
    temporary_file();
    ~temporary_file();

private:
    string file_name;
};

temporary_file::~temporary_file()
{
    // Delete the temporary file we have made
    unlink(file_name.c_str());
}
Destructors

The destructor is called automatically when the object “dies”

```c
void some_function() {
    temporary_file temp_file1;
    i++;

    if(some_condition) return;

    {
        temporary_file temp_file2;
        i++
    }

    i++;  
}

Huge benefit: you won’t forget to clean up!
```
C offers pointers, which serve a range of purposes:
- Pass large arguments to functions without copying
- Return multiple values from a function
- Efficiently return a value from a function that is partially modified
- Iterators over arrays
- Dynamic memory allocation

Pointers are difficult to deal with. Some of the above are made more robust with references.
int var;
int &ref = var; // read: int *ref = &var;

ref = ref + 1; // read: *ref = *ref + 1;

void my_function(const Person &person) {
    ...
}

... my_function(John);
But:

- You can’t change the address behind the reference
- No address arithmetic; thus no iteration
- References can’t be NULL
- Be careful not to destroy the object the reference refers to
std::vector<Person> persons;

persons.push_back(John);
Person &John_ref=persons[0];

person.push_back(Jack);
Person &Jack_ref=persons[1];
Function Overloading

- In C, many functions exist in variants for different parameter types
  - E.g.: abs, cabs, fabs, labs, llabs, fabsl, fabsf
- They all have the same intent but require separate implementations
- We don’t usually benefit from the reminder about the argument type, and it’s extra work to keep this in sync
- Wouldn’t it be nice if we could just write abs(x)? And have the compiler figure out which implementation to use?
Function Overloading

```c
int my_abs(int);
long int my_abs(long int);
float my_abs(float);
double my_abs(double);
...

int my_power(int x, int p);
float my_power(float x, float p);
...
```
Function Overloading

- Recall which basic types are really different

- If there is no perfect match, the compiler will use a scoring system

- Sorry, you can’t make the distinction using the return value

- Overloading works with methods just as well
Operator Overloading

Certain operators can be overloaded:

MyMatrix  operator+(MyMatrix, MyMatrix);
MyMatrix  operator-(MyMatrix, MyMatrix);
MyMatrix  operator*(MyMatrix, MyMatrix);
...

MyMatrix  A, B, C;

A = B * C;

For instance, this enables transparent implementation of complex numbers.
Virtual Methods

class base {
    void my_method() { std::cout << "BASE!\n"; }
};

class derived: public base {
    void my_method() { std::cout << "DERIVED!\n"; }
};

void do_something(base &b) {
    b.my_method();
}

int main() {
    derived D;
    do_something(D);
}
Virtual Methods

- The fix is to add the keyword `virtual` to the method declaration

- The primary purpose of virtual methods is typing for APIs

- Typically realised with function pointers, but all done for you
Virtual Destructors

- Virtual constructors don’t make sense. Why?

- Virtual destructors make so much sense that compilers will warn if you don’t add one, even if blank.

  Why?
Run-time error handling is a very common idiom

Usually done via `if(something bad) return -1;` or the like

It’s easy to forget, clutters code, and means that we have to have “normal data” and “error data”

Instead, throw an exception!
class my_error: public exception {
  virtual const char* what() const {
    return "my_error_message";
  }
};

... if(something bad) throw my_error();

Neat thing: cleans up any objects on the stack
Exceptions

```java
try {
    ...
}
catch (exception e) {
    // deal with it!
}
```
Templates

- We have seen templates, e.g., `std::string` or `std::vector`

- These are built using *templates*, which allow parameterizing classes, functions and typedefs with types.

- The primary purpose is library construction. Before engaging, check whether someone has already built that library you are thinking about.