Part V

Big C++

Object oriented programming (OOP)

- **Non-OOP:** functions and data that are accessible from everywhere.
- **OOP:** Data and functions (methods) together in an object.

Object oriented programming (OOP)

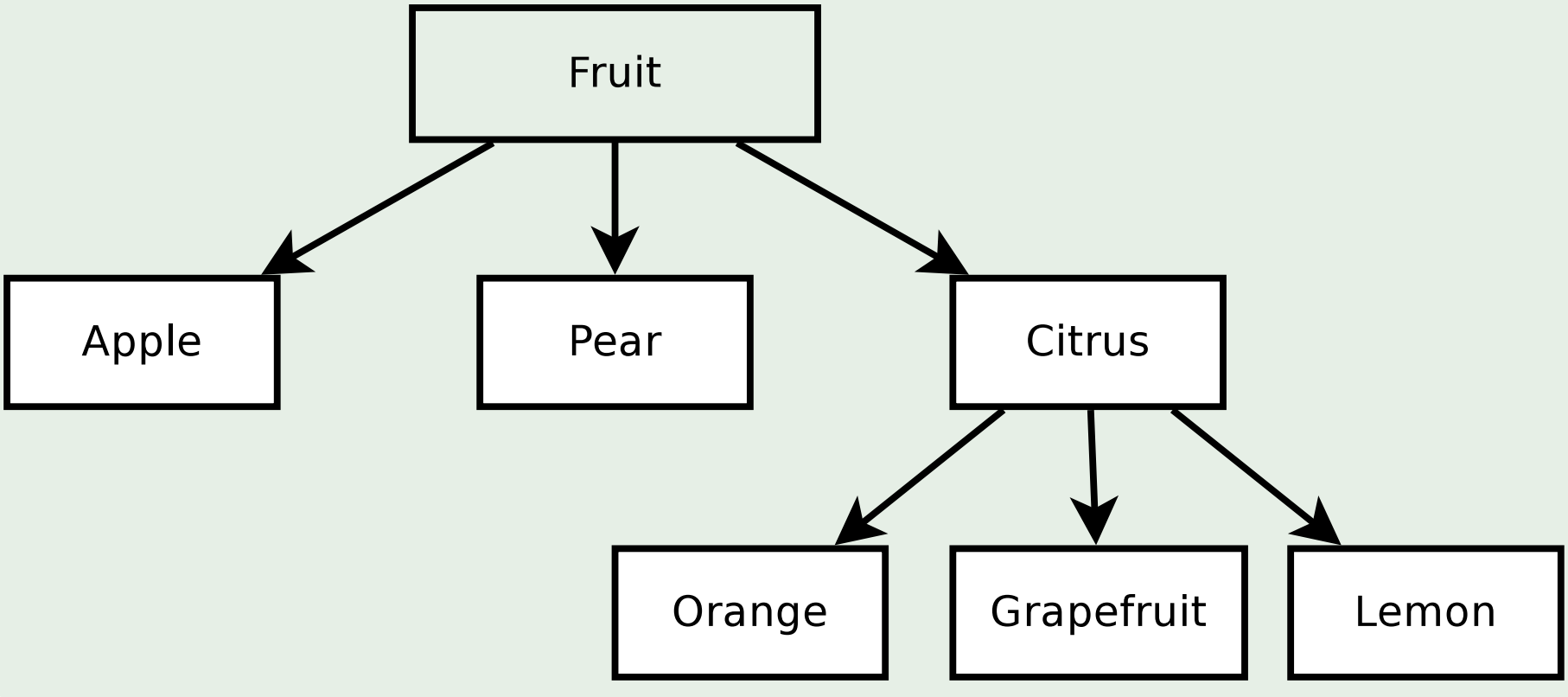
- Data is encapsulated and accessed using methods specific for that (kind of) data.
- The interface (collection of methods) should be designed around the meaning of the actions: abstraction.
- Programs typically contain multiple objects of the same type, called instances.

Object oriented programming (OOP)

- Programs typically contain different types of objects.
- Types of objects can be related, and their methods may act in the same ways, such that the same code can act on different types of object, without knowing the type: polymorphism.
- Types of object may build upon other types through inheritance.

OOP Example

Example (abstract object hierarchy)



OOP Languages

- C++ was one of the earlier languages which supported OOP. (it also supports other programming paradigms.)
- Not the earliest OOP language though: Simula, Smalltalk
- Java, C#, D all came later.
- And one can program in an object oriented fashion in almost any modern programming language (see matrix example in C).

Aspects of OOP in C++

1. Classes and objects
2. Polymorphism
3. Derived types/Inheritance
4. Advanced: Generic programming/Templates

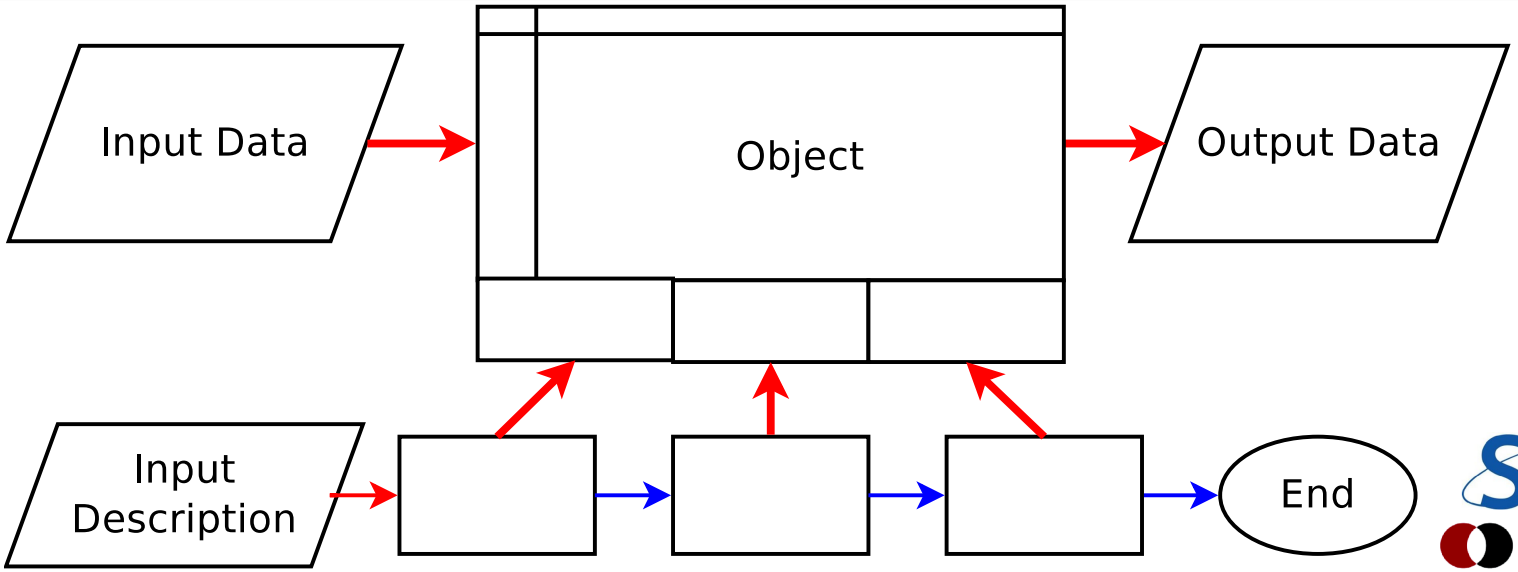
SciNet compute • calcul CANADA

Big C++: Classes and objects

What are classes and objects?

- Objects in C++ are made using 'classes'.
- A class is a type of object.
- Using a class, one can create one or more instances of that class.
- These are the objects.

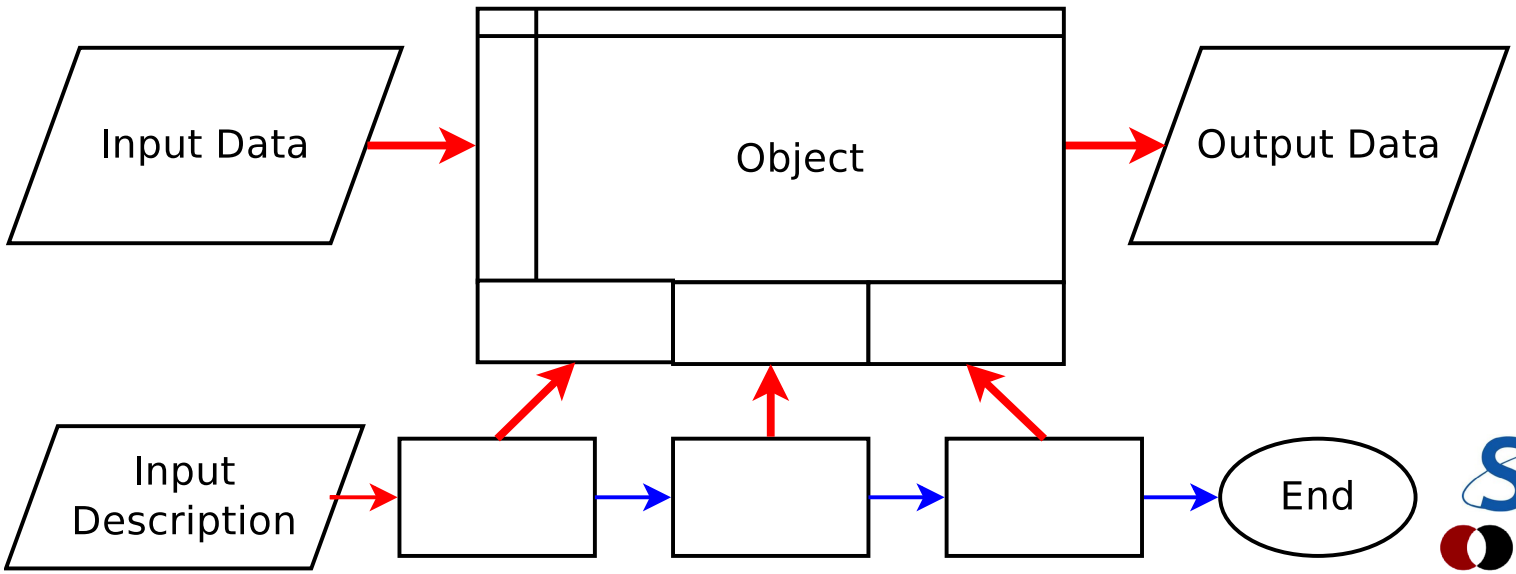
Syntactically, classes are structs with member functions.



Classes: Why structs with member functions?

An object should have properties.

1. A struct can already collect properties of different types.
2. It should be possible to declare several objects of the same type, just as in "`int x,y;`". A struct already constitutes a type definition.
3. Functions on structs often pass a pointer to a struct as a parameter. Embedding functions in structs gives a natural implied parameter.



Classes: How do we add these member functions?

```
class classname {
   public:
      type1 name1;
      type2 name2;
      type3 name3(arguments); // declare member function
      ...
};
```

- `public` allows use of members from outside the class (later more)

Example

```
class Point2D {
   public:
      int j;
      double x,y;
      void set(int aj,double ax,double ay);
};
```

Classes: How do we define these member functions?

The scope operator ::

```
type3 classname::name3(arguments) {
   statements
}
```

Example

```
void Point2D::set(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
```

Classes: How do we use the class?

Definition

```
classname objectname;
classname* objectptrname = new classname;
```

Access operator . and ->

```
objectname.name                    // variable access
objectname.name(arguments);        // member function access
objectptrname->name                // variable access
objectptrname->name(arguments); // member function access
```

Example

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.j << std::endl;
```

Classes: How do we use the class?

The `this` pointer

- The member functions of a class know what object to work on because under the hood, they are passed the pointer to the object.
- For those cases where the pointer to the object is needed, its name is always **`this`**.
- In other words, in the **`set`** function, **`j`** and **`this->j`** are the same.
- **`this`** is implicitly the first argument to the member function (this will become important in operator overloading later).

Data hiding: The secret agenda of classes

- Good components hide implementation data and member functions.
- Each member function or data member can be
  1. **private:** only member functions of the same class can access these
  2. **public:** accessible from anywhere
  3. **protected:** only this class and its derived classes have access.
- These are specified as sections within the class.

Example (Declaration)

```
class Point2D {
   private:
      int j;
      double x,y;
   public:
      void set(int aj,double ax,double ay);
      int get_j();
      double get_x();
      double get_y();
};
```

Data hiding: The secret agenda of classes

Example (Definition)

```
int Point2D::get_j() {
   return j;
}
double Point2D::get_x() {
   return x;
}
double Point2D::get_y() {
   return y;
}
```

Example (Usage)

```
Point2D myobject;
myobject.set(1,-0.5,3.14);
std::cout << myobject.get_j() << std::endl;
```

Data hiding: The secret agenda of classes

Gotcha:

When hiding the data through these kinds on accessor functions, now, each time the data is needed, a function has to be called, and there's an overhead associate with that.

- The overhead of calling this function can sometimes be optimized away by the compiler, but often it cannot.
- Considering making data is that is needed often by an algorithm just `public`, or use a `friend` .

Constructors and deconstructors

- A class defines a type, and when an instance of that type is declared, memory is allocated for that struct.
- A class is more than just a chunk of memory.
  For example, arrays may have to be allocated (`new` ...) when the object is created.
- When the object ceases to exist, some clean-up may be required (`delete` ...).

Constructor

...is called when an object is created.

Destructor

...is called when an object is destroyed.

Constructors

Declare constructors as member functions of the class with no return type:

```
class classname{
      ...
   public:
      classname(arguments);
      ...
}
```

Define them in the usual way,

```
classname::classname(arguments) {
   statements
}
```

Use them by defining an object or with new.

```
classname object(arguments);
classname* object = new classname(arguments);
```

- You usually want a constructor without arguments as well.

Constructors

Example

```
class Point2D {
   private:
      int j;
      double x,y;
   public:
      Point2D(int aj,double ax,double ay);
      int get_j();
      double get_x();
      double get_y();
};
Point2D::Point2D(int aj,double ax,double ay) {
   j = aj;
   x = ax;
   y = ay;
}
Point2D myobject(1,-0.5,3.14);
```

Destructors

Destructor

. . . is called when an object is destroyed.
Occurs when a non-static object goes out-of-scope, or when `delete` is used.
Good opportunity to release memory.

Example

```
classname* object = new classname(arguments);
...
delete object;// object deleted:  calls destructor
```

```
{
   classname object;
}// object goes out of scope:  calls destructor
```

compute • calcul CANADA

Destructors

Declare destructor as a member functions of the class with no return type, with a name which is the class name plus a ~ attached to the left.

```
class classname{
      ...
   public:
      ~classname();
      ...
}
```

Define a destructor as follows:

```
classname::~classname() {
   statements
}
```

- A destructor cannot have arguments.

Gotcha: Mixing new/delete and malloc/free

- Trivial objects (plain structs without constructors) can in principle be a allocated with `new` or with **`malloc`**.
- But pointers allocated with `new` cannot be freed using **`free`**, and for pointers allocated with **`malloc`**, `delete` should not be used.
- Non-trivial objects cannot be allocated with **`malloc`**, since the constructor is not called.
- It is best to stick to `new` and `delete` .

SciNet compute • calcul CANADA

More member functions

. . . to support the class as a new type:

1. Default constructor

   The default constructor is a constructor without arguments.
   If you have no constructors at all, C++ already knows what to do upon construction with no arguments (i.e., nothing), and you do not need to supply a default constructor (but it can still be a good idea).
   If you have any constructors with arguments, omitting a default constructor severely limits the use of the class.

2. Copy constructor

3. Assignment operator

   If the constructor allocates memory, the latter two should be supplied. If there is no memory allocation in the constructor, C++ can generate the copy constructor and assignment operator for you, performing a bit-wise or shallow copy.

Default constructor

Declaration

```
classname {
      ...
   public:
      classname();
      ...
};
```

Definition

```
classname::classname() {
   statements
}
```

This function is needed to be able to

- Declare an object without parameters: `classname name;`
- Declare an array of objects: `name = new classname[number];`

Should set elements to values so that destruction or assignment work.

Copy constructor

Declaration

```
classname {
      ...
   public:
      classname (classname & anobject);
      ...
};
```

Definition

```
classname::classname(classname & anobject) {
   statements
}
```

Used to

- Define an object using another object: `classname name(existing);`
- Pass an object by value to a function (often a bad idea).
- Return an object from a function.

Assignment operator

Declaration

```
classname {
      ...
   public:
      classname& operator=(classname & anobject);
      ...
};
```

Definition

```
classname& classname::operator=(classname & anobject) {
   statements
   return *this;
}
```

- Used to assign one object to another object: `name = existing;`
- But not in `classname name = existing;` calls the copy constructor.
- Returns a reference to this, to allow for the common C-construction

  `name = anothername = existing;`

HANDS-ON:

Convert the matrix structure to a proper c++ class, and rewrite main to use it.

Hands-on 2 - instructions

Copy the whole example_nice directory to example_big

```
$ cp -r example_nice example_big
```

Modify the code to use:

1. Classes instead of structs
2. Member functions
3. Constructors and deconstructors
4. Private member variables

Test that the code builds and runs.

Hands-on 2 - answers

If you did not quite get there, or if you have a few remaining bugs:

- Copy the c++ version I made from the **`example_big`** directory in **`scinetcppexamples.tgz`**, so we can continue later.
- Test that the code builds and runs.
- Be sure to look at the source code and see if it make sense to you.