C++ for Particle Physicists

Oliver Schulz

oschulz@mpp.mpg.de

May 21, 2014

Part I

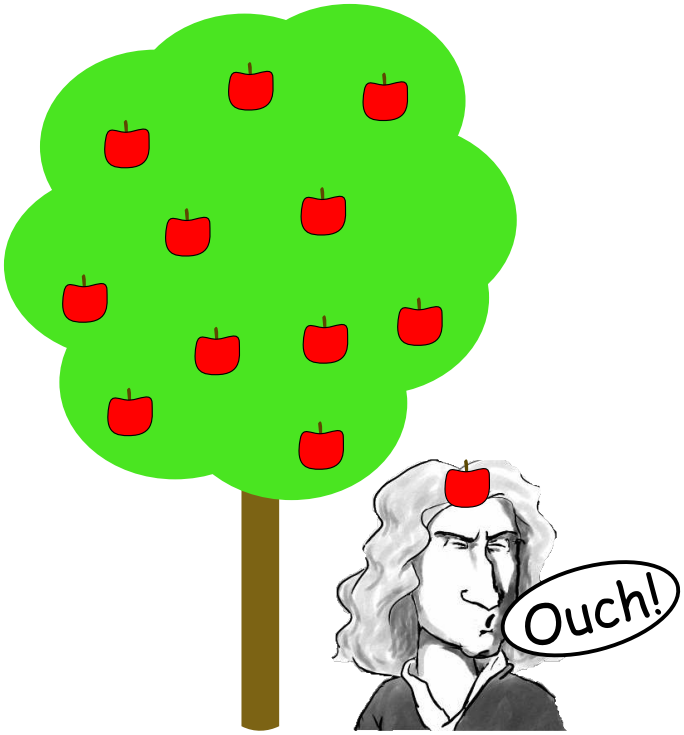
The Apple doesn't Fall Far from the Tree

About This Course

This Course

- aims to teach the basics of the C++ programming language
- also covers some tools relevant for serious C++ software development
- starts from scratch
- will (have to) skip out several more advanced aspects of C++
- wants to enable you to gain deeper understanding on your own later on
- is built around an example scenario

A Well Known Scenario ...



Object-Oriented Modelling and Programming

- complex reality
  - Entities
    (Newton, physicists, coffee machines, ...),
  - Which interact
    (talk, listen, press button, ...)
- Approximation by object oriented modelling
  - Objects (represent entities) with
    - Attributes
      (properties/information → representation of internal state)
    - Methods
      (capabilities → functions of an object)
  - Classification of alike objects

Newton as an example for a object

- Attributes
  - birthday, address, nationality, ...
  - sleeps, unconcious, ...
- Methods
  - calculateAge, ...
  - hitOnHead
- Which do we need? $\rightarrow$ relevant aspects (simplify!)

Simply put (in our context) Newton is a physicist,
who we can hit on the head (with an apple).

- Class physicist
- Method hitOnHead

C++ Syntax: Class Definitions

```
class Physicist {
public:
    void hitOnHead();
};
```

- `class` - Key word: Start of class definition
- `Physicist` - chosen, unique class name
- `{ }` - Block: delimits class body
- `hitOnHead` - chosen method name
- `()` - empty parameterlist
- `;` - marks end of statement
- `void` - this method does not return a value
- `public:` - marks beginning of externally visible members of class.

C++ Syntax: Method Definitions

```
#include <iostream>

class Physicist {
public:
    void hitOnHead();
};


void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}
```

- `class Physicist ...` - the class definition
- with declaration of method `hitOnHead()`
- definition of method `hitOnHead()` outside of class definition
- Indication of scope using `Physicist::`
- Output of text string and a newline
- using object `cout`, defined in namespace `std`.
- `#include <iostream>` "imports" pre-defined functionality, e.g. `std::cout`

Got Class! - Object?

- We have:
  - Class `Physicist`
  - Method `hitOnHead()` mit
  - Implementation of the method
- We lack:
  - A start of the program - how to get things rolling?
  - Newton as an object
  - A trigger for the hit on his head, somehow

Having defined and implemented `class Physicist` we now can use it:

```
int main() {
    Physicist newton;

    newton.hitOnHead();

    return 0;
}
```

- The `main()` function is always executed first in a C++ program (special role)
- It should return an `int` (an integer value)
- Declaration of variable `newton` of type `Physicist`
- Cool! Our first object is alive ...
- ... and we instantly hit him on the head
- Call of method `hitOnHead()` of object `newton`
- `return 0` make function `main()` return integer value `0`

The Complete Programm

```
#include <iostream>

class Physicist {
public:
    void hitOnHead();
};


void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}


int main() {
    Physicist newton;

    newton.hitOnHead();

    return 0;
}
```

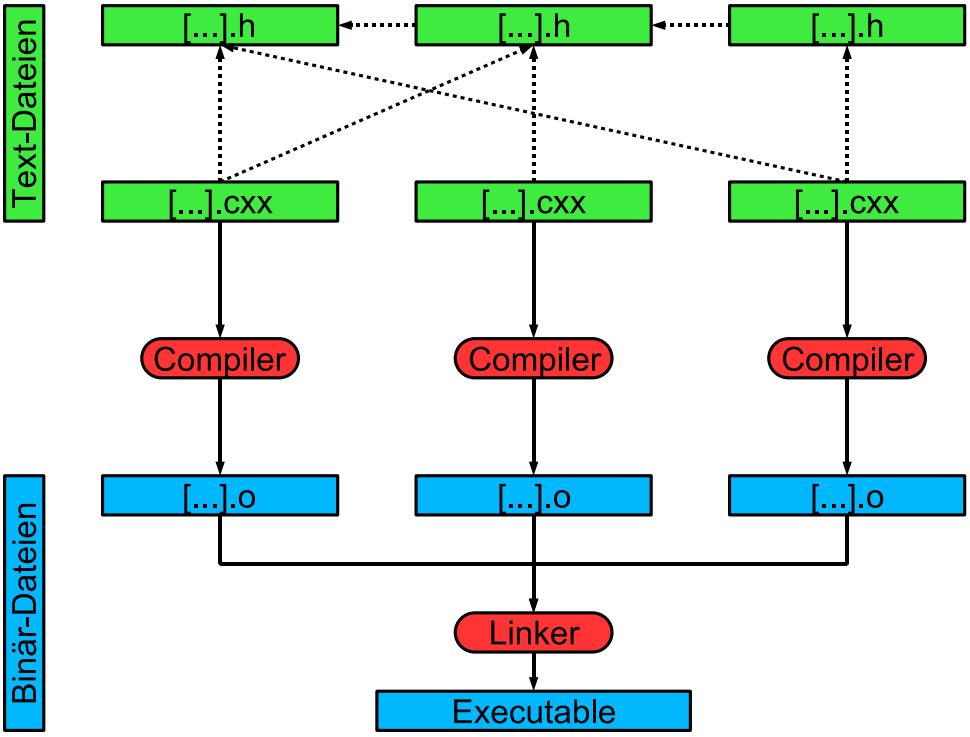
Primitive Data Types

- C++ has several build-in data types, e.g.
- `int`: Integer numbers
- `unsigned int`: positive integers
- `char`: Characters
- `float`: Floating-point numbers (real numbers)
- Primitive types are
  - low-level, based on typical CPU architectures
  - no classes, have no methods

Got program - now what?

- Source code can't be directly executed on computer CPU.
- In general two possibilities: Interpreter or compiler.
- Interpreter:
  - Programm which understands our code,
    and triggers corresponding actions on computer
  - Consumes computing time - slow
  - No intermediate steps - comfortable and flexibel
- Compiler:
  - Program that translates our source code
    into instruction set of processor $\rightarrow$ executable
  - Translation only occurs once - fast execution afterwards
  - Translation may be complex and time consuming
  - executable can run only on specific computer architecture
- Third possibility:
  Combine the two $\rightarrow$ just-in-time (JIT) compiler

Compiler und Linker



Hands on!

- Open console / terminal:

```
# g++ -c newton-sim.cxx -o newton-sim.o
# g++ newton-sim.o -o newton-sim
# ./newton-sim
```

- In simple cases: All-in-one

```
# g++ newton-sim.cxx -o newton-sim
# ./newton-sim
```

- For frequent use: All in one line

```
# g++ newton-sim.cxx -o newton-sim && ./newton-sim
```

More classes?

- Class `Physicist` - already done.
- Class `Apple`
- Class `AppleTree`
- Class `Garden` - newton and the tree have to be located somewhere after all

```
#include <iostream>

class Physicist {
public:
    void hitOnHead();
};

class Apple {
};

class AppleTree {
public:
    Apple* shake();
};

class Garden {
protected:
    Physicist newton;
    AppleTree tree;
public:
    void earthquake();
};
```

```
void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}

Apple* AppleTree::shake() {
    return new Apple;
}

void Garden::earthquake() {
    Apple *apple = tree.shake();
    if (apple != 0) {
        newton.hitOnHead();
        delete apple;
    }
}

int main () {
    Garden newtonsGarden;
    newtonsGarden.earthquake();

    return 0;
}
```

Why so complex?

Wouldn't that have been much easier?

just-ouch.cxx

```
#include <iostream>

int main () {
    std::cout « "Ouch!" « std::endl;

    return 0;
}
```

- Indeed - simple things can in principle be coded without object oriented model
- But: Object oriented approach will enable us to extend the program step by step in a modular fashion.

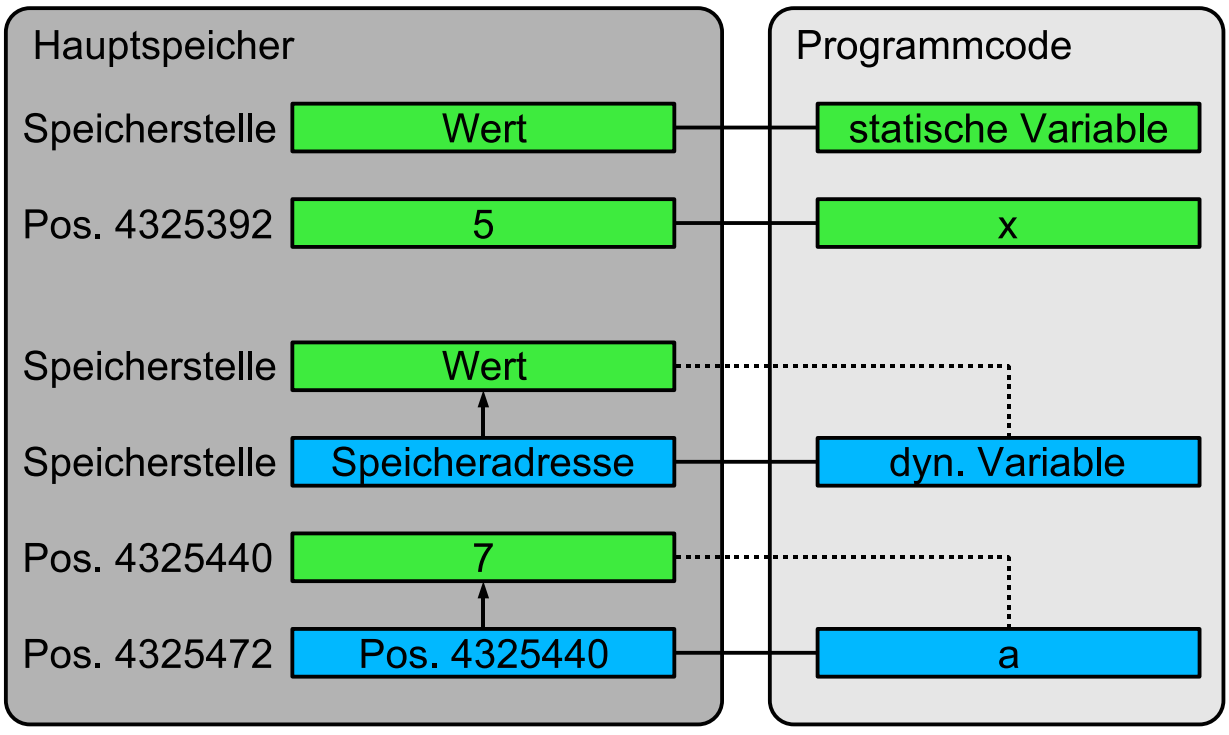
```
#include <iostream>

class Physicist {
public:
    void hitOnHead();
};

class Apple {
};

class AppleTree {
public:
    Apple* shake();
};

class Garden {
protected:
    Physicist newton;
    AppleTree tree;
public:
    void earthquake();
};
```

```
void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}

Apple* AppleTree::shake() {
    return new Apple;
}

void Garden::earthquake() {
    Apple *apple = tree.shake();
    if (apple != 0) {
        newton.hitOnHead();
        delete apple;
    }
}

int main () {
    Garden newtonsGarden;
    newtonsGarden.earthquake();

    return 0;
}
```

Static and Dynamic Variables

- Two types of variables: Static and dynamic
- Static variables:
  ```
  int i = 1;
  ```
  - Content exists only within enclosing {}-block.
- Dynamic variables:
  ```
  int *a = new int;
  *a = 4;
  ```
  - Content exists until destroyed using `delete`.
- Dynamic variables are also called pointers.

Variables and the computer memory (simplified)



Static and Dynamic Variables

- Two types of variables: Static and dynamic
- Static variables:
  `int i = 1;`
  - Content exists only within enclosing {}-block.
- Dynamic variables:
  ```
  int *a = new int;
  *a = 4;
  ```
  - Content exists until destroyed using `delete`.
- Dynamic variables are also called pointers.

static-dynamic.cxx

```
#include <iostream>

int main () {
    int i = 1;
    int j = i;
    i = 5;

    std::cout « i « std::endl;
    std::cout « j « std::endl;

    int *a = new int;
    *a = 4;
    int *b = a;
    *a = 42;

    std::cout « *a « std::endl;
    std::cout « *b « std::endl;

    delete a;

    return 0;
}
```

Pointers: The WWW as an Analogy

- Variable a is analog to a bookmark
  - 2-Tuple consisting of label (a) and URL
  - URL can be changed without changing label
- Memory location of a contains analog to URL: `http://www.google.com`
  - Points to HTML document on the web
  - Can also be pointed to by other bookmarks
- Actual content is the HTML document itself
  - contains the data
  - can be changed without changing its address

```
#include <iostream>

class Physicist {
public:
    void hitOnHead();
};

class Apple {
};

class AppleTree {
public:
    Apple* shake();
};

class Garden {
protected:
    Physicist newton;
    AppleTree tree;
public:
    void earthquake();
};
```

```
void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}

Apple* AppleTree::shake() {
    return new Apple;
}

void Garden::earthquake() {
    Apple *apple = tree.shake();
    if (apple != 0) {
        newton.hitOnHead();
        delete apple;
    }
}

int main () {
    Garden newtonsGarden;
    newtonsGarden.earthquake();

    return 0;
}
```

Overview?

- By now, things don't fit onto the screen well anymore ...
- So: Split code into separate files
- Good style - one File per class:
  `Physicist.cxx`, `Apple.cxx`, `AppleTree.cxx`, `Garden.cxx`
  `main()` remains in `newton-sim.cxx`
- Compile files one by one - and then?
- Linker can combine multiple object files
- Problem: Inter-dependencies between files
- Solution: Put code accessed by multiple files into so-called header-files and include where necessary.
- Probem: What if a header files is included several times over different paths?
- Solution: `ifndef` construct in header files

Physicist.h

```
#ifndef Physicist_h
#define Physicist_h

class Physicist {
public:
    void hitOnHead();
};

#endif // Physicist_h
```

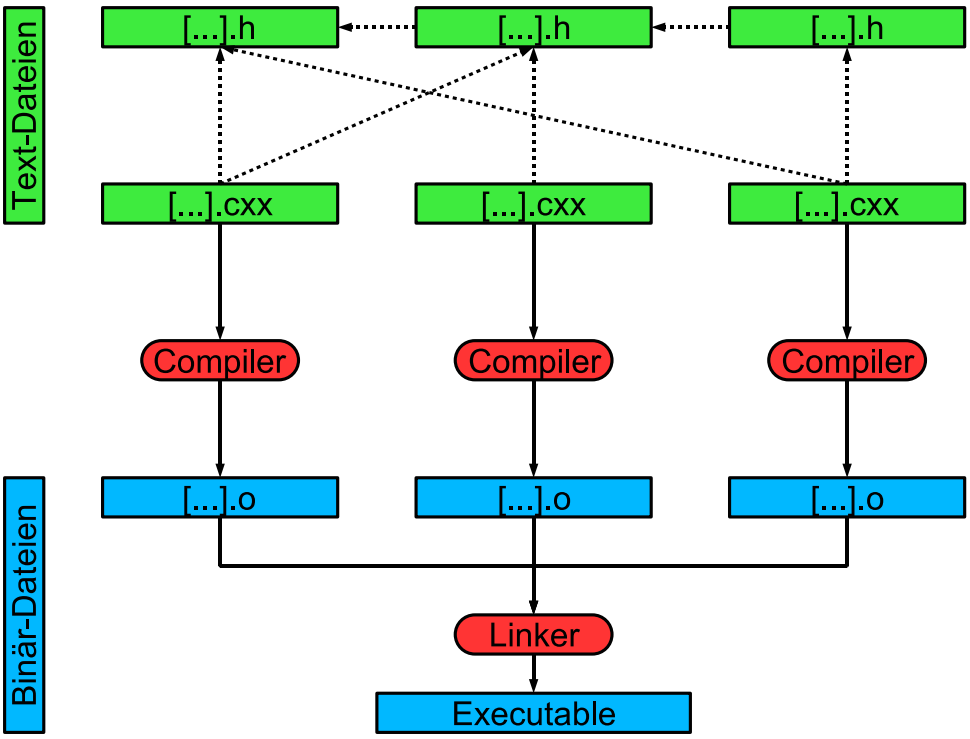
Physicist.cxx

```
#include "Physicist.h"

#include <iostream>


void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}
```

Overview?

- By now, things don't fit onto the screen well anymore ...
- So: Split code into separate files
- Good style - one File per class:
  `Physicist.cxx`, `Apple.cxx`, `AppleTree.cxx`, `Garden.cxx`
  `main()` remains in `newton-sim.cxx`
- Compile files one by one - and then?
- Linker can combine multiple object files
- Problem: Inter-dependencies between files
- Solution: Put code accessed by multiple files into so-called header-files and include where necessary.
- Probem: What if a header files is included several times over different paths?
- Solution: `ifndef` construct in header files

Compiler und Linker



Overview?

- By now, things don't fit onto the screen well anymore ...
- So: Split code into separate files
- Good style - one File per class:
  `Physicist.cxx`, `Apple.cxx`, `AppleTree.cxx`, `Garden.cxx`
  `main()` remains in `newton-sim.cxx`
- Compile files one by one - and then?
- Linker can combine multiple object files
- Problem: Inter-dependencies between files
- Solution: Put code accessed by multiple files into so-called header-files and include where necessary.
- Probem: What if a header files is included several times over different paths?
- Solution: `ifndef` construct in header files

Physicist.h

```
#ifndef Physicist_h
#define Physicist_h

class Physicist {
public:
    void hitOnHead();
};

#endif // Physicist_h
```

Physicist.cxx

```
#include "Physicist.h"

#include <iostream>


void Physicist::hitOnHead() {
    std::cout « "Ouch!" « std::endl;
}
```

Now just compile it all ...

Console

```
# g++ -c newton-sim.cxx -o newton-sim.o
# g++ -c Apple.cxx -o Apple.o
# g++ -c AppleTree.cxx -o AppleTree.o
# g++ -c Garden.cxx -o Garden.o
# g++ -c Physicist.cxx -o Physicist.o
# g++ -o newton-sim newton-sim.o Apple.o AppleTree.o Garden.o Physicist.o
```

- That looks like work - isn't there a better way to do this?
- Wouldn't it be nice to have a tool which:
  - Compiles (only) what has to be re-compiled after changes
  - Links afterwards
- Is there something like that? Of course - programmers are lazy!

Make - make it happen

- Make: System to make files ("targets") based on other files ("dependencies")
- Necessary:
  - Rules how to make the new files → `Makefile`
  - Tool which processes these rules → `make`
- Not limited to C++ beschränkt, very versatile, e.g. for LaTeX.

Make Rules

- make interprets rules of the form

Makefile

```
target: dependency1 dep2 dep3 ...
    shell-command something
    shell-command something else
```

- Take care with the indentation!
- Useful: Special wildcards and variables
  - % = Wildcard (arbitrary string)
  - $@ = target
  - $< = first depencency
  - $+ = all dependencies

A first Makefile

Makefile.simple

```
.SUFFIXES:

all: newton-sim

clean:
    rm -f *.o newton-sim

%.o : %.cxx
    g++ -c $< -o $@

newton-sim : newton-sim.o Apple.o AppleTree.o Garden.o Physicist.o
    g++ -o $@ $+

Apple.o: Apple.cxx Apple.h
AppleTree.o: AppleTree.cxx AppleTree.h Apple.h
Garden.o: Garden.cxx Garden.h Physicist.h AppleTree.h Apple.h
Physicist.o: Physicist.cxx Physicist.h
newton-sim.o: newton-sim.cxx Garden.h Physicist.h AppleTree.h Apple.h
```

Make it so!

- Aufruf von `make`:

Console

```
# make [target]
```

- Target is optional. If not specified, make will build `make` special target `all`.

A first Makefile

Makefile.simple

```
.SUFFIXES:

all: newton-sim

clean:
    rm -f *.o newton-sim

%.o : %.cxx
    g++ -c $< -o $@

newton-sim : newton-sim.o Apple.o AppleTree.o Garden.o Physicist.o
    g++ -o $@ $+

Apple.o: Apple.cxx Apple.h
AppleTree.o: AppleTree.cxx AppleTree.h Apple.h
Garden.o: Garden.cxx Garden.h Physicist.h AppleTree.h Apple.h
Physicist.o: Physicist.cxx Physicist.h
newton-sim.o: newton-sim.cxx Garden.h Physicist.h AppleTree.h Apple.h
```

Make it easier

- Again, too much work:

```
Apple.o: Apple.cxx Apple.h
AppleTree.o: AppleTree.cxx AppleTree.h Apple.h
Garden.o: Garden.cxx Garden.h Physicist.h AppleTree.h Apple.h
Physicist.o: Physicist.cxx Physicist.h
newton-sim.o: newton-sim.cxx Garden.h Physicist.h AppleTree.h Apple.h
```

- Can we automize this?

Console

```
# g++ -MM *.cxx
```

- Has to go into the `Makefile`!

Improved Makefile

Makefile

```
# Makefile for project newton-sim

.SUFFIXES:

all: newton-sim

clean:
 Makefile.dep *.o newton-sim

%.o : %.cxx
    g++ -c $< -o $@

newton-sim : newton-sim.o Apple.o AppleTree.o Garden.o Physicist.o
    g++ -o $@ $+

Makefile.dep:  *.cxx
    g++ -MM $+ > $@

include Makefile.dep
```

4. Extending the simulation there's a lot to do!

Now we want to:

- Hit Newton on the head with an apple which,
- has a fall height and a mass
- and therefore also a well-defined energy
- hit Newton multiple times
- do a few calculations on the way

Hit Him with an Apple or: Methods with Parameters

Physicist.h

```
#ifndef Physicist_h
#define Physicist_h

#include "Apple.h"


class Physicist {
public:
    void hitOnHead(Apple *apple);
};


#endif // Physicist_h
```

Method with Parameter

Physicist.cxx

```
#include "Physicist.h"

#include <iostream>


void Physicist::hitOnHead(Apple *apple) {


        std::cout « "Ouch!" « std::endl;


}
```

The apple needs more functionality

```
#ifndef Apple_h
#define Apple_h

class Apple {
protected:
  float m_mass;
  float m_energy;

public:
  float getMass() const {
    return m_mass;
  }

  float getEnergy() const {
    return m_energy;
  }

  Apple();

  Apple(float height);
};

#endif // Apple_h
```

```
#include "Apple.h"

Apple::Apple() {
  m_mass = 0.15;
  m_energy = 0;
}

Apple::Apple(float height) {
  m_mass = 0.15;
  m_energy = getMass() * height * 9.81;
}
```

New so far:

- Inline-methods (definition together with declaration)
- Keyword `const` for methods which do not change state
- Constructors
  - have the same name as their class
  - there can be several with different parameter lists
  - are executed when object is created (e.g. using `new`)
- simple calculation operators are: +,-,*,/

Method with Parameter

Physicist.cxx

```
#include "Physicist.h"

#include <iostream>


void Physicist::hitOnHead(Apple *apple) {
    float energy = apple->getEnergy();

    if (energy > 7.0) {
        std::cout « "Ouch!" « std::endl;
    };

    std::cout « energy « std::endl;
}
```

New:

- `apple->getEnergy();`
  is equivalent to:
  `(*apple).getEnergy();`
- Meaning:
  - Get object referenced by the pointer `apple` (dereferencing)
  - Call method `getEnergy()` on that object
- Arrow-notation is an abbreviation (syntactic sugar)
- Increases code-clarity when methods return pointer to object on which another methods is to be called.

Example - equivalent expressions:

```
(*(*obj_pointer_1).obj_method()).obj2_method();
obj_pointer1->obj_method()->obj2_method();
```

An Apple Up High

AppleTree.cxx

```
#include "AppleTree.h"

#include <cstdlib>

Apple* AppleTree::shake() {
    float dropHeight = 2.0 + 2.0 * float(random()) / float(RAND_MAX);

    return new Apple(dropHeight);
}
```

- Variable `dropHeight` of type `float` (real number)
- `random()` returns an integer x, so that: $0 \leq x \leq$ `RAND_MAX`
- Type-cast `float(value)` returns `value` as float
- `random()` is defined in `<cstdlib>`
- use new constructor with argument of `Apple`

4. Extending the simulation there's a lot to do!

Now we want to:

- Hit Newton on the head with an apple which,
- has a fall height and a mass
- and therefore also a well-defined energy
- hit Newton multiple times
- do a few calculations on the way

Hit 'im Again!

Garden.cxx

```
#include "Garden.h"


void Garden::earthquake() {
    for (int i=0; i<10; ++i) {
        Apple *apple = tree.shake();
        if (apple != 0) {
            newton.hitOnHead(apple);
            delete apple;
        }
    }
}
```

- `hitOnHead` now requires pointer to an apple as parameter
- Delimit block of instructions to be repeated
- `for` looping construct

The first loop

The for Statement

```
for (int i=0; i<10; ++i) {
    list of instructions
}
```

- Keyword `for` with three parameters
- `int i=0`: definition and initialization of counting variable (here: `i`)
  (executed once, before entering loop)
- `i<10`: Looping condition
  (execution *before* every every execution of loop body)
- `++i`: Increase statement (equivalent to: `i = i+1`)
  (execution *after* every every execution of loop body)
- Loop body, block `{ }` or single instruction

What's new?

- Methods with parameters: `Physicist::hitOnHead(Apple *apple)`
- Inline-methods, are declared and defined in one (in header file)
- Keyword `const` for methods, prevents object modification
- Constructors for objects: `Apple::Apple()`
- Method call notation for object pointers:
  - `apple->getEnergy();`
  - `(*apple).getEnergy();`
- Type casting: `float(random())`
- `for` loop

Do it yourself ...

Physicist.h

```
#include "Apple.h"

class Physicist {
public:
    void hitOnHead(Apple *apple);
};
```

Physicist.cxx

```
void Physicist::hitOnHead(Apple *apple) {
    float energy = apple->getEnergy();

    if (energy > 7.0) {
        std::cout « "Ouch!" « std::endl;
    };

    std::cout « energy « std::endl;
}
```

AppleTree.cxx

```
#include <cstdlib>

Apple* AppleTree::shake() {
    float dropHeight =
     2.0+2.0*float(random())/float(RAND_MAX);

    return new Apple(dropHeight);
}
```

Garden.cxx

```
void Garden::earthquake() {
    for (int i=0; i<10; ++i) {
        Apple *apple = tree.shake();
        if (apple != 0) {
            newton.hitOnHead(apple);
            delete apple;
        }
    }
}
```

The apple needs more functionality

```
#ifndef Apple_h
#define Apple_h

class Apple {
protected:
  float m_mass;
  float m_energy;

public:
  float getMass() const {
    return m_mass;
  }

  float getEnergy() const {
    return m_energy;
  }

  Apple();

  Apple(float height);
};

#endif // Apple_h
```

```
#include "Apple.h"

Apple::Apple() {
  m_mass = 0.15;
  m_energy = 0;
}

Apple::Apple(float height) {
  m_mass = 0.15;
  m_energy = getMass() * height * 9.81;
}
```

A Little Background On ifndef

- Pre-processor inserts contents of includes into source-code
- Use `#ifndef ...` and `#define ...` to prevent multiple insertions
- To test, comment ifndef (etc.) in `Apple.h` out, then run:

bash

```
# g++ -E -P Garden.cxx
```

Comments - Be Kind to the Reader!

- Comments in source code helpful and important
- Two syntax variants for comments in C++:

```
/* Every C++ Programe must define main().
   main() must return an int().

   This is a comment: The Compiler ignores everything
   between slash-star and star-slash.
*/

int main() {
    // This is a comment, too.
    // The compiler ignores everything from a double-slash to the
    // end of line.

    return 0;  // No errors, so return 0.
}
```

Interface Documentation

- At some point, simple comments in source code are not enough:
  Want browseable documentation as HTML (or PDF, or ...)
- Idea: Document classes, methods and attributes directly in source code using formal scheme
- Why isn't there a system that automatically generates ... ?
- There is (in fact, there are several)

Apple.h

```
...
/// @brief A garden variety apple.
///
/// Mmmmmm, juicy!

class Apple {
protected:
    float m_mass;   /// This apple's mass.
    float m_energy; /// Internal energetic state of the apple.

public:
    /// @brief  Get this apple's mass.
    /// @return A mass (in kg).
    ///
    /// Not much more to say ...
    float getMass() const { return m_mass; }
...
    /// @brief  Create an Apple in free fall.
    /// @param  height  The height to create the Apple at.
    ///
    /// The Apple will begin to fall immediately!
    Apple(float height);
};
...
```

Doxygen

- Doxygen: System to generatre API (application program interface)
  documentation from special source code comments
- Needs config file `Doxyfile`, create with:

```
# doxygen -g
```

- Ok, let's go:

```
# doxygen
```

- Let's put it in the `Makefile`
  (and add a few other improvements)

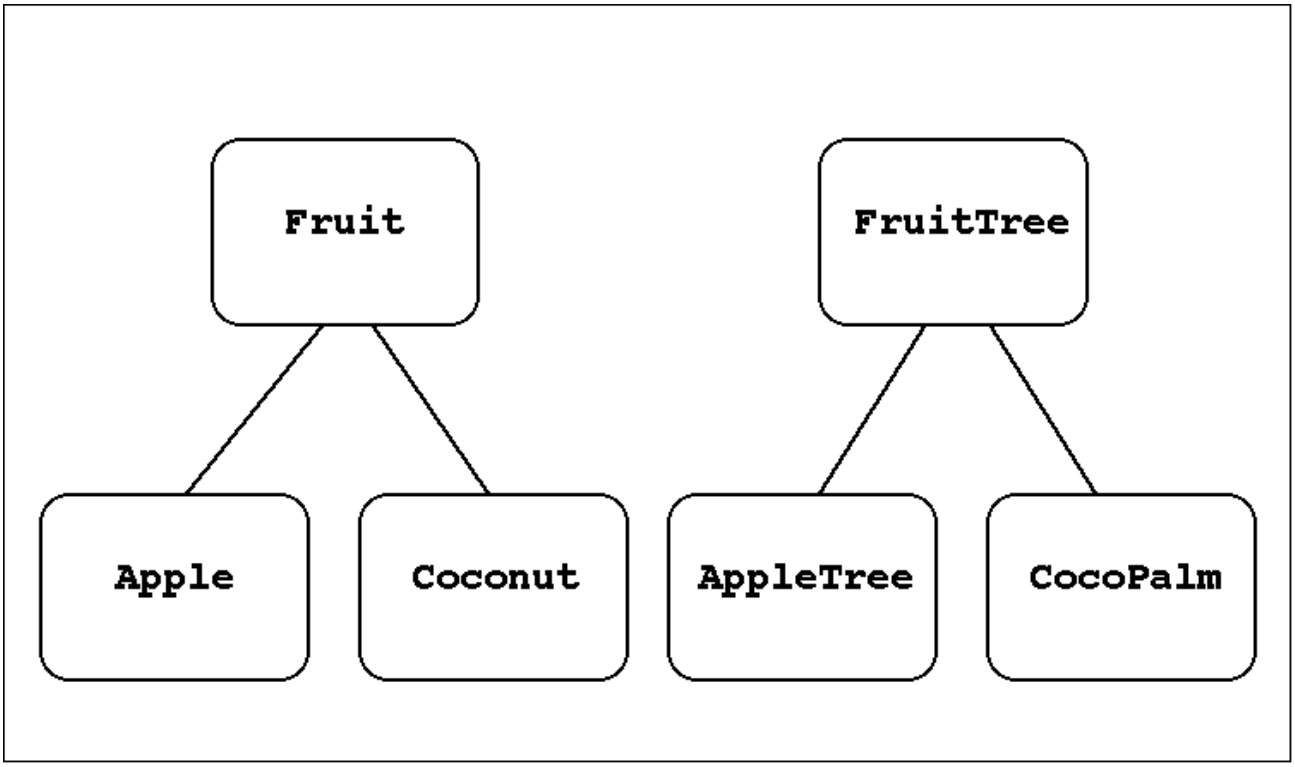
```
# Makefile for project newton-sim

.SUFFIXES:

all: newton-sim

doc:
    doxygen

clean:
    $(RM) Makefile.dep *.o newton-sim

maintainer-clean: clean
    $(RM) -r html latex

%.o : %.cxx
    $(CXX) -c $< -o $@

newton-sim : newton-sim.o Apple.o AppleTree.o Garden.o Physicist.o
    $(CXX) -o $@ $+

Makefile.dep: *.cxx
    $(CXX) -MM $+ > $@

include Makefile.dep
```

Inheritance for Better Structure and Less Work

- Modelling (Classification, ...)
- Unite classes with similar structure
  - common properties are true subset of methods and attributes
- Move common structure into common parent class (super class)

→ Child classes inherit common structure

Inheritance: An Example



Apple.h

```
class Apple : public Fruit {
protected:
    float m_mass;
    float m_energy;

public:
    float getMass() const {
        return m_mass; }
    float getEnergy() const {
        return m_ernergy; }

    Apple();

    Apple(float height);
};
```

Fruit.h

```
class Fruit {
protected:
    float m_mass;
    float m_energy;

public:
    virtual float getMass() const {
        return m_mass; }
    float getEnergy() const {
        return m_ernergy; }

    Fruit();

    Fruit(float mass, float height);

    virtual ~Fruit();
};
```

Inheritance so far:

- `class Apple : public Fruit {}`
  inherits from super class (herer `Apple` from `Fruit`)
- keyword `virtual`
  enables override of method behaviour in child class

Also new:

- `class virtual ~Fruit();`
  destructor as complement to constructor

Apple.h

```
class Apple : public Fruit {
public:
    Apple(float height);
};
```

Apple.cxx

```
#include "Apple.h"

#include <iostream>

using namespace std;

Apple::Apple(float height) :
  Fruit(0.15, height) {
    cout «
    "Apple: I seem to be falling ..."
    « endl;
}
```

Initializers:

- `Apple::Apple(float height) : Fruit(0.15, height)`
  specification of initializer (constructor of `Fruit`)

Using namespaces:

- `using namespace std;`
  - imports a namespace (`std` here)
    so we don't have to use explicit scoping all the time
  - *Never* use in header files! Never ever!

Coconut.h

```
class Coconut : public Fruit {
public:
    Coconut(float height);

    virtual ~Coconut();
};
```

Coconut.cxx

```
using namespace std;

Coconut::Coconut(float height):Fruit(2.0,height){
  cout « "Coconut: Here we go ..." « endl;
}

Coconut::~Coconut() {
    cout « "Coconut: Thud." « endl;
}
```

AppleTree.h

```
class AppleTree : public FruitTree {
public:
    virtual Fruit* shake();
};
```

FruitTree.h

```
class FruitTree {
protected:
    static float rndValue() {
        return float(random()) /
               float(RAND_MAX);
    }

    static float randomDropHeight();

public:
    virtual Fruit* shake() = 0;

    virtual ~FruitTree();
};
```

Vererbung:

- Keyword `static`
  : Method is bound to class, not specific instances (objects)
- `virtual Fruit* shake() = 0;`
  declares an abstract method (no implementation in this class)
  - abstract method makes whole class abstract
  - abstract class cannot be instantiated
  - child classes are also abstract
    until all abstract methods are implemented

FruitTree.h

```
class FruitTree {
protected:
    static float rndValue() {
        return float(random()) /
               float(RAND_MAX);
    }

    static float randomDropHeight();

public:
    virtual Fruit* shake() = 0;

    virtual ~FruitTree();
};
```

FruitTree.cxx

```
float FruitTree::randomDropHeight() {

    float dropHeight = 3.0 + (rndValue()
        +rndValue()+rndValue()+rndValue()
        +rndValue()+rndValue()) / 6 - 0.5;
    return dropHeight;
}

FruitTree::~FruitTree() {}
```

AppleTree.h

```
#ifndef AppleTree_h
#define AppleTree_h

#include "FruitTree.h"

class AppleTree : public FruitTree {
public:
    virtual Fruit* shake();
};

#endif // AppleTree_h
```

AppleTree.cxx

```
#include "AppleTree.h"
#include "Apple.h"

Fruit* AppleTree::shake() {
    return new Apple(randomDropHeight());
}
```

CocoPalm.h

```
#ifndef CocoPalm_h
#define CocoPalm_h

#include "FruitTree.h"

class CocoPalm : public FruitTree {
public:
    virtual Fruit* shake();
};

#endif // CocoPalm_h
```

CocoPalm.cxx

```
#include "CocoPalm.h"
#include "Coconut.h"

Fruit* CocoPalm::shake() {
  return rndValue() < 0.33
    ? new Coconut(randomDropHeigt())
    : 0;
}
```

Garden.h

```
#ifndef Garden_h
#define Garden_h

#include "Physicist.h"
#include "AppleTree.h"
#include "CocoPalm.h"

class Garden {
protected:
    Physicist newton;
    AppleTree tree;
    CocoPalm  palm;

public:
    void earthquake();
};

#endif // Garden_h
```

Garden.cxx

```
#include "Garden.h"
#include <iostream>

using namespace std;

void Garden::earthquake() {
  for (int i=0; i<10; ++i) {
    cout «
    "Garden: The earth is shaking!"
    « endl;

    Fruit *fruit[2] =
      {tree.shake(), palm.shake()};
    for (int i=0; i<2; ++i)
      if (fruit[i] != 0) {
        newton.hitOnHead(fruit[i]);
        delete fruit[i];
    }

        cout « endl;
    }
}
```

What's new?

- Inheritance
  - Super classes and inheriting classes
  - `virtual`
  - `static`
  - abstract methods and classes
- `using namespace`
- Ternary ? : operator
- Destructors
- `array` example

Make more

- Current Makefile doesn't support:
  installation, de-installation, distribution, etc.
- Convention: `make`-targets `install`, `uninstall`, `dist` u.a.
- So: Extend Makefile with new targets and functionality?
- Also: Compiler- and linker options and properties, tools, libraries, etc. differ between systems
- So: (Operating-)System specific Makefiles?
- Sounds like a standard problem - standard solution?
- GNU Autotools: Creation of standardized `Makefile`'s and handling of system-specific stuff

GNU Autotools

- GNU Autotools: Set of tools to create `Makefile`'s, and to do a lot more
- Philosophy: Developer uses Autotools, to create `Makefile`-prototypes und `configure`-script
  Users use `configure` to create final Makefiles and then `make` to build on target system
- Automatic detection of system-specific features
- Support to find required for programs / tools and libraries
- Automatic standard `make`-Targets like `install`, `uninstall` und `dist`.
- Targeted to software projects (especially C/C++)
  less flexible than hand-written Makefiles.

Autotools in Detail

- Workflow (simplified):
  - `aclocal`, `autoheader`, `libtoolize`, `autopoint` → misc. auxiliary files
  - `Makefile.am`'s → `automake` → `Makefile.in`'s
  - `configure.ac`'s, `Makefile.in`'s → `autoconf` → `configure`, `Makefile`'s
- Lots of tools - but don't have to call them manually: `autoreconf` does it automagically

Project File Structure Conventions

- Let's improve our project file structure.
- Sub-Directories `src` und `doc` for source code, resp. documentation (very common).
- README, general information, installation instructions, etc. in project root directory

A Recipe

We need:

- `configure.ac` in project root directory: Basis for `configure`.
- `Makefile.am` in every directory: Basis for `Makefile`'s.
- Im project root directory:
  - `AUTHORS`: Author list
  - `ChangeLog`: Changes from version to version
  - `COPYING`: Licensing information
  - `INSTALL`: Installation instructions
  - `NEWS`: Information about new features, etc.
  - `README`: Description of this project / package
  - No technical function, but common and sensible to have
- Also common: Simple shell script `autogen.sh` to call Autotools (`autoreconf`)

configure.ac

```
# Initialize:

AC_INIT([newton-sim], [0.7.0])
AM_INIT_AUTOMAKE([-Wall -Werror])

# Checks for programs:

AC_PROG_CXX

# Output:

AC_CONFIG_HEADERS([config.h])

AC_CONFIG_FILES([
    Makefile
    src/Makefile
    doc/Makefile doc/Doxyfile
])

AC_OUTPUT
```

Makefile.am

```
MAINTAINERCLEANFILES = Makefile.in \
    aclocal.m4 config.guess config.h.in config.sub configure \
    depcomp install-sh ltmain.sh missing

EXTRA_DIST = autogen.sh

SUBDIRS = src doc

dist_doc_DATA = README COPYING AUTHORS ChangeLog INSTALL NEWS
```

src/Makefile.am

```
MAINTAINERCLEANFILES = Makefile.in

bin_PROGRAMS = newton-sim

newton_sim_SOURCES = newton-sim.cxx Apple.cxx AppleTree.cxx Coconut.cxx \
    CocoPalm.cxx Fruit.cxx Garden.cxx FruitTree.cxx Physicist.cxx
```

doc/Makefile.am

```
MAINTAINERCLEANFILES = Makefile.in html/* latex/*

EXTRA_DIST = html/index.html html latex/refman.tex latex

html/index.html latex/refman.tex: $(top_srcdir)/src/*.cxx $(top_srcdir)/src/*.h
    doxygen
```

doc/Doxyfile.in

```
[...]
INPUT = @top_srcdir@/src
[...]
```

autogen.sh

```
#!/bin/sh

autoreconf -install
```

Prepare, Build, Cleap Up

- `# ./autogen.sh`: Autotools (developers only)
- `# ./configure`: `Makefile`'s erzeugen (developers and users)
- `# make`: build (developers and users)
- `# make clean`: Remove files created by `make`
- `# make distclean`: Also remove files created by `configure`
- `# make maintainer-clean`: Also remove files created by Autotools

Software Distribution and Installation

- `# make dist`: Bundle of software package (tarball, `.tar.gz`).
- `# make install`: Install software on system
- `# make uninstall`: Remove software from system

Branching

The if statement

```
if (Condition) Expression_1 else Expression_2
```

- Keyword `if` with condition
- `Condition`: Boolean expression
- `Expression_1`: Single instruction or block (`{ }`) of instructions
  (executed if `Condition == TRUE`)
- Keyword `else`: Alternative branch
  (optional)
- `Ausdruck_2`: Single instruction or block (`{ }`) of instructions
  (executed if `Condition == FALSE`)

Loops with pre-condition

The while loop

```
while (Condition) {
    ...
    Instructions
    ...
}
```

- Keyword `while` with condition
- `Condition`: Boolean expression
- `Instructions` to be repeated
  (while `Condition == TRUE`)
- If `Condition == FALSE` on entering,
  $\rightarrow$ `Instructions` will never be executed

Loops with post-condition

The do loop

```
do {
    ...
    Instructions
    ...
} while (Condition)
```

- Keyword `do` without argument
- `Instructions` to be repated
- `while` with `Condition`
- `Condition`: Boolean expression
- If `Condition == FALSE` on entering,
  → `Instructions` will still be executed once

Breaking out of loops

break

```
while (Condition) {
    some instructions ...
    if (done == TRUE) break;
    further instructions ...
}
```

- An example `while` loop
- `done`: boolean variable (used as break-out condition)
- Keyword `break`: triggers breaking out of the loop.
  (`further instructions` are not executed after break)
- Loop is terminated immediately, independent of `Condition`.

Short-Circuiting Loops

continue

```
while (Condition) {
    some instructions ...
    if (enough_for_now == TRUE) continue;
    further instructions ...
}
```

- Another example `while` loop
- `enough_for_now`: boolean variable
- Keyword `continue`: Skip all following instructions and re-enter loop.
- Before re-entering, `while`-condition is checked.

What's missing?

- A good physicist write down his observations ...
- ... surely Newton has a journal!
- Need new classes:
  - Notepad: A physicists lab notebook.
  - Hit: Notebook-entry, description of an event.

Hit.h

```
class Hit  {
public:
    enum Type {
        TP_UNKNOWN = 0,
        TP_APPLE   = 1,
        TP_COCONUT = 2
    };

protected:
    float m_time;
    Type m_type;
    float m_energy;

public:
    float getTime() const { return m_time; }

    Type getType() const { return m_type; }

    float getEnergy() const { return m_energy; }

    Hit(float time, Type type, float m_energy)
        : m_time(time), m_type(type), m_energy(m_energy) {}

    Hit() : m_time(0), m_type(TP_UNKNOWN), m_energy(0){}
};
```

Notepad.h

```
#include <list>

#include "Hit.h"

class Notepad : public std::list<Hit> {
public:
    void print() const;
};
```

Notepad.cxx

```
#include "Notepad.h"

#include <string>
#include <iostream>

using namespace std;

void Notepad::print() const {
    for (const_iterator hit=begin(); hit!=end(); ++hit) {
        string type = "U";
        if (hit->getType() == Hit::TP_APPLE) type = "A";
        else if (hit->getType() == Hit::TP_COCONUT) type = "C";

        cout « hit->getTime() « "\t" « type « "\t" « hit->getEnergy() « endl;
    }
}
```

Physicist.h

```
...
#include "Notepad.h"
...
class Physicist {
protected:
    Notepad m_notepad;
...
public:

    void hitOnHead(float time, Fruit *fruit);
    void relax();
};
```

Physicist.cxx

```
...
#include <typeinfo>

#include "Apple.h"
#include "Coconut.h"
...
void Physicist::hitOnHead(float time, Fruit *fruit) {
    float energy = fruit->getEnergy();
...
    Hit::Type type = Hit::TP_UNKNOWN;
    if (typeid(*fruit) == typeid(Apple)) type = Hit::TP_APPLE;
    else if (typeid(*fruit) == typeid(Coconut)) type = Hit::TP_COCONUT;

    m_notepad.push_back(Hit(time, type, energy));
}

void Physicist::relax() {
    m_notepad.print();
}
```

Garden.h

```
...
class Garden {
protected:
    float time;
...
public:
    void earthquake(unsigned int shakes=20);

Garden();
};
```

Garden.cxx

```
...
void Garden::earthquake(unsigned int shakes) {
    for (int i=0; i<shakes; ++i) {
        ...
        time = time + float(random()) / float(RAND_MAX);
        ...
            newton.hitOnHead(time, fruit[i]);
        ...

    newton.relax();
}

Garden::Garden() : time(1.0) {}
```

newton-sim.cxx

```
#include "Garden.h"

#include <cstdlib>

int main (int argc, char *argv[]) {
    unsigned int shakes = 5;
    if (argc >= 2) shakes = atoi(argv[1]);

    Garden newtonsGarden;
    newtonsGarden.earthquake(shakes);

    return 0;
}
```

Saving Your Measurement Values

- Redirect output to a file:
  ```
  # newton-sim > data.txt
  ```
- Problem: Output contains debug and info output
- Idea: Separate output types.
  - Measurement values → standard-output
  - Everything else → error-output
- Then we can do this:
  ```
  # newton-sim > data.txt 2> error.txt
  ```
- To write to error-output instead of standard-output, replace `std::cout` by `std::cerr`

File Paths

- Preprocessor
  - `-I/my/include/path`: Add include file path
- Linker
  - `-L/my/lib/path`: Add library search path
  - `-lmylib`: Link `libmylib.{a|so}`
- Operating System (for execution)
  - `$PATH`: Executable search path
  - Show dynamic library dependencies with `$ldd some_binary`
  - Dynamic library search path:
    - Linux: `$LD_LIBRARY_PATH`
    - OS-X: `$DYLD_LIBRARY_PATH`
    - Windows: `$PATH`

Part II

The ROOT of All Evil

The CERN ROOT System

ROOT includes:

- Extensive C++ class libraries:
  histograms, event lists, data analysis, GUI, ...
- A file format to store C++ objects on disk
- Graphical tools for data analysis
- A C++ Interpreter (CINT, replaced by CLING in ROOT-6)
- Tools to enable reflection with C++ classes
- GUI Editor, Geometry editor and viewer, and much more

Advantages and Disadvantages

ROOT is:

- powerful
- Almost a standard in particle physics

ROOT is also (sadly):

- complex
- not well documented in some important places
- not well designed
- not modular

Newton and ROOT

No let's:

- Read in the output of `newton-sim`
- Create a histogram of the hit energies
- Save the simulation output to a ROOT file
- Take a look at the output with some graphical tools
- Make `newton-sim` write directly to a ROOT file

Starting ROOT

- # root
- # root script.C
- # root 'function.C("test")'
- # root datafile.root

readascii.C

```
#include <iostream>

using namespace std;

void readascii(const std::string &srcName) {
    ifstream src;
    src.open(srcName.c_str());

    float time; char ascType; float energy;

    while (src.good()) {
        src » time » ascType » energy;
        if (!src.good()) break;
        cout « time « "\t" « ascType « "\t" « energy « endl;
    }

    src.close();
}
```

ascii2hist.C

```
...
#include <TH1F.h>

using namespace std;

TH1F* ascii2hist(const std::string &srcName) {
    TH1F* hist = new TH1F("Hits", "Hits on Newton's head", 2048, 0, 80);

    ifstream src;
    src.open(srcName.c_str());

    float time; char ascType; float energy;

    while (src.good()) {
        src » time » ascType » energy;
        if (!src.good()) break;
        cerr « time « "\t" « ascType « "\t" « energy « endl;
        hist->Fill(energy);
    }

    src.close();
    return hist;
}
```

TFiles and TTrees

- ROOT offers custom file format (.root) for C++ objects, access via class `TFile`.
- TFile save several C++ objects and have sub-directories
- Special storage class `TTree` in `TFile`:
  - Internal structure of objects forms a tree
  - `TTree` stores a list of these trees (entries)
  - Well suited for complex physics events.
    In general useful for time series data.
- Can only store instances of classes with a `Streamer`

Streamers and Dictionaries

- To store objects, ROOT has to know their internal structure (reflection).
- Class structure information is provided by "Dictionary", generated from Header files
- Dictionary is to CINT what header files are to the compiler.
- Classes with streamers or derived from `TObject` also have to include `ClassDef` macro.
- Special `pragma link` construct and ROOT-Tools for dictionary generation

Hit.h

```
...
#include <Rtypes.h>
...
class Hit {
...
    virtual ~Hit() {}

    ClassDef(Hit, 1);
};

#ifdef __CINT__
#pragma link C++ class Hit+;
#endif
...
```

Hit.cxx

```
#include "Hit.h"

ClassImp (Hit)
```

ascii2tfile.C

```
...
#include <TFile.h>
#include <TTree.h>
#include <Hit.h>

TFile *ascii2tfile(const std::string &srcName, const std::string &trgName) {
    Hit *hit = new Hit;

    TFile *tfile = new TFile(trgName.c_str(), "RECREATE");
    TTree *ttree = new TTree("SimData", "Newton Simulation Data");
    ttree->Branch("NewtonHits", "Hit", &hit);
...
    while (src.good()) {
...
        Hit::Type type=Hit::TP_UNKNOWN;
        if (ascType == 'A') type = Hit::TP_APPLE;
        else if (ascType == 'C') type = Hit::TP_COCONUT;
        *hit = Hit(time, type, energy);
        ttree->Fill();
    }

    ttree->Write();
    delete hit;
    return tfile;
}
```

Class-loading with ACLiC

- `ascii2tfile.C` doesn't run in ROOT's C++ interpreter (CINT). So we need to compile:
- ACLiC: Automatic Compiler of Libraries for CINT
- ACLiC compiles source code, generates dictionary and outputs all to a dynamic library loadable by ROOT
- Running ACLiC (from ROOT console):
  ```
  .L Hit.cxx+
  .L ascii2tfile.C+
  ```
- Plus-sign results in running ACLiC. Recompilation only when code has changed (similar to make).
- Running `ascii2tfile`:
  ```
  TFile *outFile = ascii2tfile("simdata.txt", \
  "simdata.root");
  ```
- Can we do this as a script?

convert.C

```
{

gROOT->ProcessLine(".L Hit.cxx+");
gROOT->ProcessLine(".L ascii2tfile.C+");

TFile *outFile = ascii2tfile("simdata.txt", "simdata.root");

}
```

Why not to it in one step?

- Want to create `.root`-file directly from simulation
- More control over output

Notepad.h

```
...
#include <string>
...
class Notepad : public std::list<Hit> {
public:
    void print() const;

    void write2root(const std::string& fileName) const;
 };
...
```

Notepad.cxx

```
...
#include <TFile.h>
#include <TTree.h>
...
void Notepad::write2root(const std::string& fileName) const {
    const Hit *hitPointer = 0;

    TFile *tfile = new TFile(fileName.c_str(), "RECREATE");
    TTree *ttree = new TTree("SimData", "Newton Simulation Data");
    ttree->Branch("NewtonHits", "Hit", &hitPointer);

    for (const_iterator hit=begin(); hit!=end(); ++hit) {
        hitPointer = &(*hit);
        ttree->Fill();
    }

    ttree->Write();
    delete tfile;
}
...
```

Physicist.h

```
...
public:
  void hitOnHead(float time, Fruit *fruit);
  void relax();
  const Notepad& getNotepad() const {return m_notepad;}
}
```

- Die Implementierung von `relax()` in `Physicist.cxx` fällt natürlich auch weg.

Garden.h

```
...
public:
  void earthquake(unsigned int shakes=20);

  const Physicist& getPhysicist() const {return newton;}
...
}
```

newton-sim.cxx - main

```
...
Garden newtonsGarden;
newtonsGarden.earthquake(shakes);

const Notepad &notepad = newtonsGarden.getPhysicist().getNotepad();
notepad.print();
notepad.write2root("simdata.root");
...
```

Class(ic) Acquaintances

- How to acquaintant ROOT with our classes?
- Would enable direct interaction and CINT-scripting with our own classes
- Let's try this with ACLiC ...

loadClasses.C

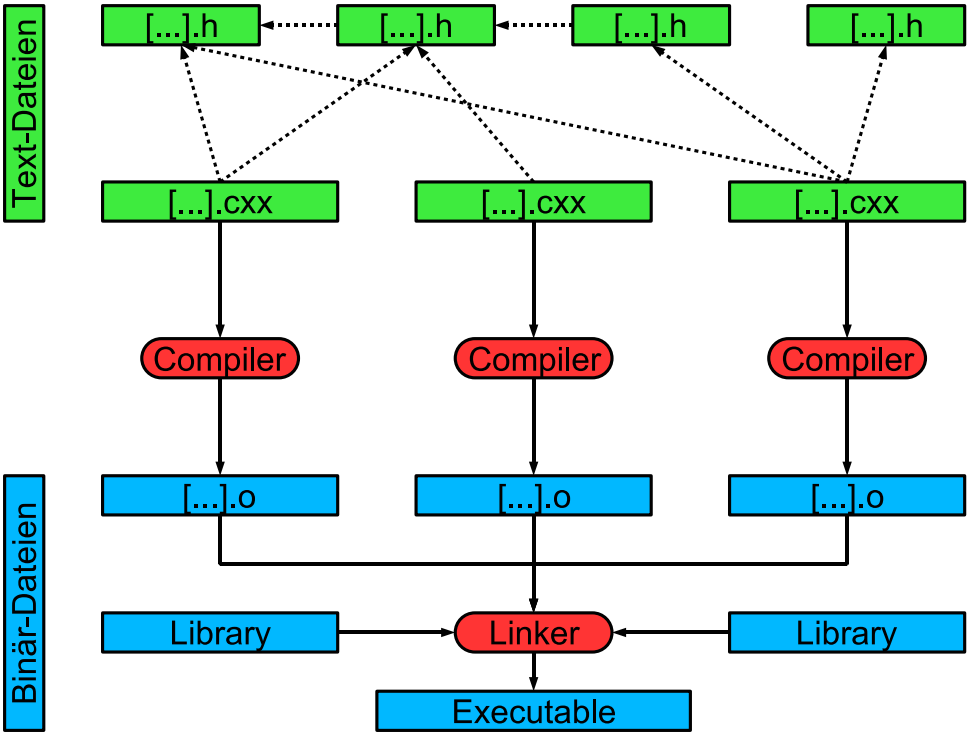
```
{

gROOT->ProcessLine(".L Fruit.cxx+");
gROOT->ProcessLine(".L FruitTree.cxx+");
gROOT->ProcessLine(".L Apple.cxx+");
gROOT->ProcessLine(".L AppleTree.cxx+");
gROOT->ProcessLine(".L Coconut.cxx+");
gROOT->ProcessLine(".L CocoPalm.cxx+");
gROOT->ProcessLine(".L Hit.cxx+");
gROOT->ProcessLine(".L Notepad.cxx+");
gROOT->ProcessLine(".L Physicist.cxx+");
gROOT->ProcessLine(".L Garden.cxx+");

gROOT->ProcessLine(".L HitSelector.cxx+");

}
```

Have to load / compile classes in right order!

Not, but ...

- `loadClasses` works, but:
- Class interdependencies make things problematic
- We already have a very nice automatic build system (Autotools)
- Can't build executables this way
- Integration of ROOT with Autotools/Make?

Compiler und Linker - Libraries



Libraries

- Libraries provide binary code for stuff from .cxx files.
- Two types: static and dynamic
  - static: Binary code embedded into executable at link time
  - dynamisch: Only reference to library added to executable at link time - executable needs to load library at run time!
- Reflection and class loading in ROOT only possible with dynamic libraries
- Need to link against ROOT libraries, also want to create our own library → modify build system.

newton-sim-LinkDef.h

```
#ifdef __CINT__
#pragma link C++ class Apple-;
#pragma link C++ class AppleTree-;
#pragma link C++ class Coconut-;
#pragma link C++ class CocoPalm-;
#pragma link C++ class Fruit-;
#pragma link C++ class FruitTree-;
#pragma link C++ class Garden-;
#pragma link C++ class Hit+;
#pragma link C++ class Notepad-;
#pragma link C++ class Physicist-;
#endif
```

prama-Statements für all classes (also move pragma for `Hit` here). Minus-sign at the end: Don't create streamer.

configure.ac

```
...
# Checks for programs:

AC_PROG_CXX
AC_LIBTOOL_DLOPEN
AC_PROG_LIBTOOL

# Checks for libraries:

AC_CHECK_PROGS(ROOTCONFIG, root-config, false])
AC_CHECK_PROGS(ROOTCINT, rootcint, false)

if test "${ROOTCONFIG}" = false; then AC_MSG_ERROR([Need root-config.]); fi
if test "${ROOTCINT}" = false; then AC_MSG_ERROR([Need rootcint.]); fi

CXXFLAGS="$CXXFLAGS `${ROOTCONFIG} --cflags`"
LIBS="`${ROOTCONFIG} --ldflags --glibs` -lSpectrum $LIBS"

# Output:
...
```

src/Makefile.am

```
CLEANFILES = *.d *.so *-rdict.cxx *-rdict.h
MAINTAINERCLEANFILES = Makefile.in

lib_LTLIBRARIES = libnewton-sim.la
libnewton_sim_la_SOURCES = libnewton-sim-rdict.cxx \
    Apple.cxx AppleTree.cxx Coconut.cxx CocoPalm.cxx Fruit.cxx Garden.cxx \
    FruitTree.cxx Hit.cxx Notepad.cxx Physicist.cxx
libnewton_sim_la_headers = \
    Apple.h AppleTree.h Coconut.h CocoPalm.h Fruit.h Garden.h \
    FruitTree.h Hit.h Notepad.h Physicist.h

include_HEADERS = $(libnewton_sim_la_headers)

bin_PROGRAMS = newton-sim
newton_sim_SOURCES = newton-sim.cxx
newton_sim_LDADD = libnewton-sim.la
# newton_sim_LDFLAGS = -static

libnewton-sim-rdict.cxx: $(libnewton_sim_la_headers) newton-sim-LinkDef.h
    $(ROOTCINT) -f $@ -c -p $(CXXFLAGS) $+

rootmapdir = $(libdir)
rootmap_DATA = .libs/libnewton-sim.rootmap
.libs/libnewton-sim.rootmap: libnewton-sim-rdict.cxx $(lib_LTLIBRARIES)
    cat $< | grep 'void .*_ShowMembers' | sed 's/.*void \(.*\)_ShowMembers.*/Library.\1:
```

What's happening here?

- Dynamic library `libnewton-sim.so` and binary executable `newton-sim` created in `src/.libs`.
- `newton-sim` is linked dynamically against `libnewton-sim.so`, will need it at run time
- `libnewton-sim.so` will contain ROOT dictionaries for classes
- Shell-script `newton-sim` will be created in `src`, calls binary executable `src/.libs/newton-sim` with correct `LD_LIBRARY_PATH` auf.
- libnewton-sim.rootmap will be created in `src/.libs`, tells ROOT, which libraries to load for which classes. Get's installed next to libraries.

ROOT + CINT

- Well suited for
  - (Interactive) use of existing classes and short scripts
- Less suited for
  - More complex applications
    (CINT not fully compatible with C++)
  - Time-critical code
  - Custom classes
- Unsuited for
  - Code that needs external (non-ROOT) libraries
  - Standalone executables
  - Installable applications

ROOT + ACLiC

- Well suited for
  - Small projects with few classes
  - Quick and dirty coding
- Less suited for
  - Many classes (cumbersome)
  - Longer-term and shared projects
  - Code that need external (non-ROOT) libraries
- Unsuited for
  - Code that needs external (non-ROOT) libraries
  - Standalone executables
  - Installable applications

ROOT + Autotools/Make

- Well suited for
  - Bigger and/or long-term projects
  - Code that needs external (non-ROOT) libraries
- Less suited for
  - Quick and dirty coding

runSelector.C

```
{

//gROOT->ProcessLine(".x loadClasses.C");
//gROOT->ProcessLine(".L .libs/libnewton-sim.so");

TFile *f = TFile::Open("simdata.root");
TTree *t = (TTree *)(f->Get("SimData"));

t->Process("HitSelector", "", 5, 0);

}
```

Part III

Advanced Topics

Templates

Templated class:

```
template <typename T, int n> class MyClass {
public:
    T v[n];
};
```

Templated function / method:

```
template <typename T> T mult(const T &a, const T &b) {
    return a * b;
}
```

Also valid:

```
template <class T> ...
```

Exceptions

```
#include <stdexcept>
#include <typeinfo>
#include <iostream>

int main() {
    try {
        // ...
        throw std::runtime_error("Some error");
        // ...
        throw std::out_of_range("Index out of range");
        // ...
        return 0;
    }
    catch(const std::runtime_error &e) {
        std::cerr « "RUNTIME ERROR: " « e.what() « std::endl;
        return 1;
    }
    catch(const std::exception &e) {
        std::cerr « "GENERIC EXCEPTION: " « typeid(e).name() « ", " « e.what() « std::endl;
        return 1;
    }
    catch(...) {
        std::cerr « "UNKNOWN EXCEPTION: " « std::endl;
        return 1;
    }
}
```

New in C++11

- Range-based for loop
- Type inference: `auto` and `decltype`
- Uniform initialization and initializer lists
- Lambda functions
- `std::unique_ptr` and `std::shared_ptr`
- Rvalue references and move semantics
- Calling contructors from other constuctors
- `nullptr` and `nullptr_t`
- And much, much more - see http://en.wikipedia.org/wiki/C++11

What we left out - and why

- Details about the preprocessor - usually not needed in C++
- Classes and objects:
  - `mutable`, `volatile` - only for low-level coding
  - `struct` - inherited from C, treat as `class` where all is public
  - `friend` - only necessary in special cases with well-planned classes
  - Multiple inheritance - usually not necessary, needs experience
- `switch-case`-Statement - easy to make mistakes, (use `if-else-if`)
- `goto` - NO GO!
- Multi-dimensional static arrays - rarely useful

Literature and Documentation

C++

- Loudon, Kyle; C++ Pocket Reference;
  O'Reilly 2003
- Stroustrup, Bjarne; The C++ Programming Language;
  Addison-Wesley 2000

Root

- http://root.cern.ch
- Most important: User's Guide, Reference Guide, tutorials