Contents

Chapter 1

Adding a Class

1.1 The Role of TObject

The light-weight **TObject** class provides the default behavior and protocol for the objects in the ROOT system. Specifically, it is the primary interface to classes providing object I/O, error handling, inspection, introspection, and drawing. The interface to this service is via abstract classes.

1.1.1 Introspection, Reflection and Run Time Type Identification

Introspection, which is also referred to as reflection, or run time type identification (RTTI) is the ability of a class to reflect upon itself or to "look inside itself. ROOT implements reflection with the **TClass** class. It provides all the information about a class, a full description of data members and methods, including the comment field and the method parameter types. A class with the `ClassDef` macro has the ability to obtain a **TClass** with the `IsA` method.

```
TClass *cl = obj->IsA();
```

It returns a **TClass**. In addition, an object can directly get the class name and the base classes by:

```
const char* name = obj->ClassName();
```

If the class is a descendent of **TObject**, you can check if an object inherits from a specific class, you can use the `InheritsFrom` method. This method returns `kTrue` if the object inherits from the specified class name or **TClass**.

```
Bool_t b = obj->InheritsFrom("TLine");
Bool_t b = obj->InheritsFrom(TLine::Class());
```

ROOT and `Cling` rely on reflection and the class dictionary to identify the type of a variable at run time. With **TObject** inheritance come some methods that use Introspection to help you see the data in the object or class. For instance:

```
obj->Dump();       // lists all data members and their current valsue
obj->Inspect();    // opens a window to browse data members
obj->DrawClass(); // Draws the class inheritance tree
```

For an example of `obj->Inspect()`, see "Inspecting Objects".

1.1.2 Collections

To store an object in a ROOT collection, it must be a descendent of **TObject**. This is convenient if you want to store objects of different classes in the same collection and execute the method of the same name on all members of the collection. For example, the list of graphics primitives are in a ROOT collection called **TList**. When the canvas is drawn, the `Paint` method is executed on the entire collection. Each member may be a different class, and if the `Paint` method is not implemented, **TObject::Paint** will be executed.

1.1.3 Input/Output

The `TObject::Write` method is the interface to the ROOT I/O system. It streams the object into a buffer using the `Streamer` method. It supports cycle numbers and automatic schema evolution. See "Input/Output".

1.1.4 Paint/Draw

These graphics methods are defaults; their implementation in **TObject** does not use the graphics subsystem. The `TObject::Draw` method is simply a call to `AppendPad`. The `Paint` method is empty. The default is provided so that one can call `Paint` in a collection. The method `GetDrawOption` returns the draw option that was used when the object was drawn on the canvas. This is especially relevant with histograms and graphs.

1.1.5 Clone/DrawClone

Two useful methods are `Clone` and `DrawClone`. The `Clone` method takes a snapshot of the object with the `Streamer` and creates a new object. The `DrawClone` method does the same thing and in addition draws the clone.

1.1.6 Browse

This method is called if the object is browse-able and is to be displayed in the object browser. For example the **TTree** implementation of `Browse`, calls the Browse method for each branch. The `TBranch::Browse` method displays the name of each leaf. For the object's `Browse` method to be called, the `IsFolder()` method must be overridden to return true. This does not mean it has to be a folder, it just means that it is browse-able.

1.1.7 SavePrimitive

This method is called by a canvas on its list of primitives, when the canvas is saved as a script. The purpose of `SavePrimitve` is to save a primitive as a C++ statement(s). Most ROOT classes implement the `SavePrimitive` method. It is recommended that the `SavePrimitive` is implemented in user defined classes if it is to be drawn on a canvas. Such that the command `TCanvas::SaveAs(Canvas.C)` will preserve the user-class object in the resulting script.

1.1.8 GetObjectInfo

This method is called when displaying the event status in a canvas. To show the event status window, select the `Options` menu and the `EventStatus` item. This method returns a string of information about the object at position (x, y). Every time the cursor moves, the object under the cursor executes the `GetObjectInfo` method. The string is then shown in the status bar. There is a default implementation in **TObject**, but it is typically overridden for classes that can report peculiarities for different cursor positions (for example the bin contents in a TH1).

1.1.9 IsFolder

By default an object inheriting from **TObject** is not brows-able, because **TObject::IsFolder()** returns `kFALSE`. To make a class browse-able, the `IsFolder` method needs to be overridden to return `kTRUE`. In general, this method returns `kTRUE` if the object contains browse-able objects (like containers or lists of other objects).

1.1.10 Bit Masks and Unique ID

A **TObject** descendent inherits two data members: `fBits` and `fUniqueID`. `fBits`is 32-bit data member used with a bit mask to get object information. Bits 0 - 13 are reserved as global bits, bits 14 - 23 can be used in different class hierarchies.

```
enum EObjBits {
   kCanDelete     = BIT(0), // if can be deleted
   kMustCleanup   = BIT(3), // if destructor must call RecursiveRemove()
   kObjInCanvas   = BIT(3), // for backward compatibility only
   kIsReferenced  = BIT(4), // if referenced by TRef or TRefArray
   kHasUUID       = BIT(5), // if has a TUUID, fUniqueID=UUIDNumber
   kCannotPick    = BIT(6), // if cannot be picked in a pad
   kNoContextMenu = BIT(8), // if does not want a context menu
   kInvalidObject = BIT(13) // object ctor succeeded but the object should not be used
};
```

For example, the bits `kMustCleanup` and `kCanDelete` are used in **`TObject`**. See "The kCanDelete Bit" and "The kMustCleanup Bit". They can be set by any object and should not be reused. Make sure to no overlap in any given hierarchy them. The bit 13 (`kInvalidObject`) is set when an object could not be read from a ROOT file. It will check this bit and will skip to the next object on the file.

The **`TObject`** constructor initializes the `fBits` to zero depending if the object is created on the stack or allocated on the heap. When the object is created on the stack, the `kCanDelete` bit is set to false to protect from deleting objects on the stack. The high 8 bits are reserved for the system usage; the low 24 bits are user settable. `fUniqueID` is a data member used to give a unique identification number to an object. It is initialized to zero by the **`TObject`** constructor. ROOT does not use this data member. The two data members (`fBits` and `fUniqueID`) are streamed out when writing an object to disk. If you do not use them, you can save some space and time by specifying:

```
MyClass::Class()->IgnoreTObjectStreamer();
```

This sets a bit in the **`TClass`** object. If the file is compressed, the savings are minimal since most values are zero; however, it saves some space when the file is not compressed. A call to`IgnoreTObjectStreamer` also prevents the creation of two additional branches when splitting the object. If left alone, two branches called `fBits` and `fUniqueID` will appear.

1.2 Motivation

If you want to integrate and use your classes with ROOT, to enjoy features like, extensive RTTI (Run Time Type Information) and ROOT object I/O and inspection, you have to add the following line to your class header files:

```
ClassDef(ClassName,ClassVersionID);  //The class title
```

For example in `TLine.h` we have:

```
ClassDef(TLine,1);                   //A line segment
```

The **`ClassVersionID`** is used by the ROOT I/O system. It is written on the output stream and during reading you can check this version ID and take appropriate action depending on the value of the ID. See "Streamers". Every time you change the data members of a class, you should increase its `ClassVersionID` by one. The `ClassVersionID` should be `>=1`. Set `ClassVersionID=0` in case you don't need object I/O. To be able to generate properly documentation for your classes using **`THtml`** you must add the statement:

```
ClassImp(ClassName)
```

For example in `TLine.cxx`:

```
ClassImp(TLine)
```

Note that you should provide a default constructor for your classes, i.e. a constructor with zero parameters or with one or more parameters all with default values in case you want to use object I/O. If do not provide such a default contructor, you MUST implement an I/O constructor. If not you will get a compile time error. See the "The Default Constructor" paragraph in this chapter. The `ClassDef` and `ClassImp` macros are defined in the file `Rtypes.h`. This file is referenced by all ROOT include files, so you will automatically get them if you use a ROOT include file.

1.2.1 Template Support

In ROOT version 3.03 and older, ROOT provided special `ClassDef` and `ClassImp` macros for classes with two and three template arguments. In ROOT version 3.04 and above, the macros `ClassDef` and `ClassImp` can be used directly even for a class template. `ClassImp` is used to register an implementation file in a class. For class templates, the `ClassImp` can only be used for a specific class template instance.

```
ClassImp(MyClass1<double>);
```

For multiple template arguments, you will need to use an intermediary `typedef`:

```
typedef MyClass2<int,float> myc_i_f;
ClassImp(myc_i_f);
```

You can also register an implementation for all instances of a class template by using `templateClassImp`:

```
templateClassImp(MyClass3);
```

Here are examples of a header and a `LinkDef` file:

```
// in header file MyClass.h
template <typename T> class MyClass1 {
private:
      T  fA;
      ...
public:
      ...
      ClassDef(MyClass1,1)
};
template <typename T1, typename T2> class MyClass2 {
private:
      T1  fA;
      T2  fB;
public:
      ...
      ClassDef(MyClass2,1)
};
template <typename T1, typename T2, typename T3> class MyClass3 {
private:
      T1  fA;
      T2  fB;
      T3  fC;
      ...
public:
      ...
      ClassDef(MyClass3,1)
};
```

```
// A LinkDef.h file with all the explicit template instances
// that will be needed at link time
#ifdef __CLING__

#pragma link C++ class MyClass1<float>+;
#pragma link C++ class MyClass1<double>+;
#pragma link C++ class MyClass2<float,int>+;
#pragma link C++ class MyClass2<float,double>+;
#pragma link C++ class MyClass3<float,int,TObject*>+;
#pragma link C++ class MyClass3<float,TEvent*,TObject*>+;

#endif
```

1.3 The Default Constructor

ROOT object I/O requires every class to have either a default constructor or an I/O constructor. A default constructor is a constructor with zero parameters or with one or more parameters all with default values. An I/O constructor is a constructor with exactly one parameter which type is a pointer to one of the type marked as an 'io constructor type'. We will come back to this context in a few paragraphs. This default or I/O constructor is called whenever an object is being read from a ROOT database. Be sure that you do not allocate any space for embedded pointer objects in this constructor. This space will be lost (memory leak) while reading in the object. For example:

```
class T49Event : public TObject {
private:
Int_t        fId;
TCollection *fTracks;
...
public:
   // Error space for TList pointer will be lost
T49Event() { fId = 0; fTrack = new TList; }
   // Correct default initialization of pointer
T49Event() { fId = 0; fTrack = 0; }
...
};
```

The memory will be lost because during reading of the object the pointer will be set to the object it was pointing to at the time the object was written. Create the `fTrack` list when you need it, e.g. when you start filling the list or in a **not-default** constructor.

```
...
if (!fTrack) fTrack = new TList;
...
```

The constructor actually called by the ROOT I/O can be customized by using the rootcling pragma:

```
#pragma link C++ ioctortype UserClass;
```

For example, with this pragma and a class named MyClass, the ROOT I/O will call the first of the following 3 constructors which exists and is public:

```
MyClass(UserClass*);MyClass(TRootIOCtor*);
MyClass();    // Or a constructor with all its arguments defaulted.
```

When more than one pragma ioctortype is used, the first seen as priority. For example with:

```
#pragma link C++ ioctortype UserClass1;
#pragma link C++ ioctortype UserClass2;
```

We look for the first existing public constructor in the following order:

```
MyClass(UserClass1*);
MyClass(UserClass2*);
MyClass(TRootIOCtor*);
MyClass();   // Or a constructor with all its arguments defaulted.
```

1.4 rootcling: The Cling Dictionary Generator

A way in which dictionaries can be generated is via the `rootcling` utility. This tool generates takes as input a set of headers and generates in output the dictionary C++ code and a `pcm` file. This latter file is fundamental for the correct functioning of the dictionary at runtime. It should be located in the directory where the shared library is installed in which the compiled dictionary resides.

NOTA BENE: the dictionaries that will be used within the same project must have unique names. In other words, compiled object files relative to dictionary source files cannot reside in the same library or in two libraries loaded by the same application if the original source files have the same name. This loose limitation is imposed by the registration mechanism ROOT has in place to keep track of dynamically loaded libraries.

In the following example, we walk through the steps necessary to generate a dictionary, I/O, and inspect member functions. Let's start with a **`TEvent`** class, which contains a collection of **`TTracks`**.

The `TEvent.h` header is:

```
#ifndef __TEvent__
#define __TEvent__
#include "TObject.h"
class TCollection;
class TTrack;

class TEvent : public TObject {
private:
Int_t        fId;          // event sequential id
Float_t      fTotalMom;    // total momentum
TCollection *fTracks;      // collection of tracks
public:
TEvent() { fId = 0; fTracks = 0; }
TEvent(Int_t id);
~TEvent();
void    AddTrack(TTrack *t);
Int_t   GetId() const { return fId; }
Int_t   GetNoTracks() const;
void    Print(Option_t *opt="");
Float_t TotalMomentum();
```

```
ClassDef(TEvent,1);  //Simple event class
};
```

The things to notice in these header files are:

- The usage of the `ClassDef` macro
- The default constructors of the **`TEvent`** and **`TTrack`** classes
- Comments to describe the data members and the comment after the `ClassDef` macro to describe the class

These classes are intended for you to create an event object with a certain id, and then add tracks to it. The track objects have a pointer to their event. This shows that the I/O system correctly handles circular references.

The `TTrack.h` header is:

```
#ifndef __TTrack__
#define __TTrack__
#include "TObject.h"

class TEvent;
class TTrack : public TObject {
   private:
      Int_t    fId;       //track sequential id
      TEvent  *fEvent;    //event to which track belongs
      Float_t  fPx;       //x part of track momentum
      Float_t  fPy;       //y part of track momentum
      Float_t  fPz;       //z part of track momentum
   public:
      TTrack() { fId = 0; fEvent = 0; fPx = fPy = fPz = 0; }
      TTrack(Int_t id, Event *ev, Float_t px,Float_t py,Float_t pz);
      Float_t  Momentum() const;
      TEvent  *GetEvent() const { return fEvent; }
      void     Print(Option_t *opt="");

   ClassDef (TTrack,1);  //Simple track class
};

#endif
```

Next is the implementation of these two classes.

`TEvent.cxx`:

```
#include <iostream.h>

#include "TOrdCollection.h"
#include "TEvent.h"
#include "TTrack.h"

ClassImp(TEvent)

...
```

`TTrack.cxx`:

```
#include <iostream.h>

#include "TMath.h"
#include "Track.h"
#include "Event.h"

ClassImp(TTrack)
...
```

Now using `rootcling` we can generate the dictionary file.

```
rootcling eventdict.cxx -c TEvent.h TTrack.h
```

Looking in the file `eventdict.cxx` we can see, the`Streamer()` and`ShowMembers()`methods for the two classes. `Streamer()` is used to stream an object to/from a **`TBuffer`** and `ShowMembers()` is used by the `Dump()` and `Inspect()` methods of **`TObject`**. Here is the `TEvent::Streamer` method:

```
void TEvent::Streamer(TBuffer &R__b) {
   // Stream an object of class TEvent.
   if (R__b.IsReading()) {
      Version_t R__v = R__b.ReadVersion();
      TObject::(R__b);
      R__b >> fId;
      R__b >> fTotalMom;
      R__b >> fTracks;
   } else {
      R__b.WriteVersion(TEvent::IsA());
      TObject::Streamer(R__b);
      R__b << fId;
      R__b << fTotalMom;
      R__b << fTracks;
   }
}
```

The **`TBuffer`** class overloads the `operator<<()` and `operator>>()` for all basic types and for pointers to objects. These operators write and read from the buffer and take care of any needed byte swapping to make the buffer machine independent. During writing, the **`TBuffer`** keeps track of the objects that have been written and multiple references to the same object are replaced by an index. In addition, the object's class information is stored. **`TEvent`** and **`TTracks`** need manual intervention. Cut and paste the generated `Streamer()` from the `eventdict.cxx` into the class' source file and modify as needed (e.g. add counter for array of basic types) and disable the generation of the `Streamer()` when using the `LinkDef.h` file for next execution of `rootcling`. In case you do not want to read or write this class (no I/O) you can tell `rootcling` to generate a dummy `Streamer()`by changing this line in the source file:

```
ClassDef(TEvent,0);
```

If you want to prevent the generation of `Streamer()`, see the section "Adding a Class with a Shared Library".

1.4.1 Dictionaries for STL

Usually, headers are passed to rootcling at the command line. To generate a dictionary for a class from the STL, e.g.

std::vector<MyClass>, you would normally pass the header defining **MyClass** and **std::vector**. The latter is a compiler specific header and cannot be passed to rootcling directly. Instead, create a little header file that includes both headers, and pass that to rootcling.

Often ROOT knows where **MyClass** and the templated class (e.g. vector) are defined, for example because the files got **#included**. Knowing these header files ROOT can automatically generate the dictionary for any template combination (e.g. **vector<myClass>**) when it is needed, by generating files starting with **AutoDict***. You can toggle this feature on or off at the ROOT prompt by executing **.autodict**.

1.5 Adding a Class with a Shared Library

Step 1: Define your own class in `SClass.h` and implement it in `SClass.cxx`. You must provide a default constructor or an I/O constructor for your class. See the "The Default Constructor" paragraph in this chapter.

```
#include <iostream.h>
#include "TObject.h"
class SClass : public TObject {
   private:
      Float_t   fX;         //x position in centimeters
      Float_t   fY;         //y position in centimeters
      Int_t     fTempValue; //! temporary state value
   public:
      SClass()            { fX = fY = -1; }
      void Print() const;
      void SetX(float x) { fX = x; }
      void SetY(float y) { fY = y; }
```

```
ClassDef(SClass, 1)
};
```

Step 2: Add a call to the `ClassDef` macro to at the end of the class definition (in the `SClass.h` file). `ClassDef(SClass,1)`. Add a call to the `ClassImp` macro in the implementation file (`SClass.cxx`): `ClassImp(SClass)`.

```
// SClass.cxx:
#include "SClass.h"
ClassImp(SClass);
void SClass::Print() const {
   cout << "fX = " << fX << ", fY = " << fY << endl;
}
```

You can add a class without using the `ClassDef` and `ClassImp` macros; however, you will be limited. Specifically the object I/O features of ROOT will not be available to you for these classes. See "Cling the C++ Interpreter". The `ShowMembers` and `Streamer` method, as well as the `>>` operator overloads, are implemented only if you use `ClassDef` and `ClassImp`. See `$ROOTSYS/include/Rtypes.h` for the definition of `ClassDef` and `ClassImp`. To exclude a data member from the `Streamer`, add a `!` as the first character in the comments of the field:

```
Int_t     fTempValue; //! temporary state value
```

1.5.1 The LinkDef.h File

Step 3: The `LinkDef.h` file tells `rootcling` which classes should be added to the dictionary.

```
#ifdef __CLING__
#pragma link C++ class SClass;
#endif
```

Three options can trail the class name:

- `-` : tells `rootcling` **not** to generate the `Streamer` method for this class. This is necessary for those classes that need a customized `Streamer` method.

```
#pragma link C++ class SClass-;  // no streamer
```

- `!` : tells `rootcling` **not** to generate the `operator>>(TBuffer &b,MyClass *&obj)` method for this class. This is necessary to be able to write pointers to objects of classes not inheriting from `TObject`.

```
#pragma link C++ class SClass!;      // no >> operator
// or
#pragma link C++ class SClass-!; // no streamer, no >> operator
```

- `+` : in ROOT version 1 and 2 tells `rootcling` to generate a `Streamer` with extra byte count information. This adds an integer to each object in the output buffer, but it allows for powerful error correction in case a `Streamer` method is out of sync with data in the file. The `+` option is mutual exclusive with both the `-` and `!` options.

IMPORTANT NOTE: In ROOT Version 3 and later, a "+" after the class name tells `rootcling` to use the new I/O system. The byte count check is always added. The new I/O system has many advantages including support automatic schema evolution, full support for STL collections and better run-time performance. We strongly recommend using it.

```
#pragma link C++ class SClass+; // add byte count
```

For information on `Streamers` see "Input/Output". To get help on `rootcling` type on the UNIX command line: `rootcling -h`

1.5.1.1 The Order Matters

When using template classes, the order of the pragma statements matters. For example, here is a template class `Tmpl` and a normal class `Norm`, which holds a specialized instance of a `Tmpl`:

```
class Norm {
private:
   Tmpl<int>* fIntTmpl;
public:
  ...
};
```

Then in `Linkdef.h`, the pragma statements must be ordered by listing all specializations before any classes that need them:

```
// Correct Linkdef.h ordering
...
#pragma link C++ class Tmpl<int>;
#pragma link C++ class Norm;
...
```

And not vice versa:

```
// Bad Linkdef.h ordering
...
#pragma link C++ class Norm;
#pragma link C++ class Tmpl<int>;
...
```

In this case, `rootcling` generates `Norm::Streamer()` that makes reference to `Tmpl<int>::Streamer()`. Then `rootcling` gets to process `Tmpl<int>` and generates a specialized `Tmpl<int>::Streamer()` function. The problem is, when the compiler finds the first `Tmpl<int>::Streamer()`, it will instantiate it. However, later in the file it finds the specialized version that `rootcling` generated. This causes the error. However, if the `Linkdef.h` order is reversed then `rootcling` can generate the specialized `Tmpl<int>::Streamer()` before it is needed (and thus never instantiated by the compiler).

1.5.1.2 Other Useful Pragma Statements

The complete list of pragma statements currently supported by Cling is:

```
#pragma link [C|C++|off] all [class|function|global|typedef];
#pragma link [C|C++|off]
   [class|struct|union|enum|namespace|protected][name];
#pragma link [C|C++|off] [global|typedef][name];
#pragma link [C|C++|off] [nestedclass|nestedtypedef];

#pragma link [C++|C|off|MACRO] function [name]<(argtypes)>;
#pragma link
   [C++|C|off|MACRO] function [classname]::[name]<(argtypes)>;
#pragma link off all methods;
#pragma link [C|C++|off] defined_in [filename];
#pragma link
   [C|C++|off] defined_in [class|struct|namespace] [name];
#pragma link [C|C++|off] all_function   [classname];
#pragma link [C|C++|off] all_datamember [classname];
```

The `[classname]` and the `[name]` can also contain wildcards. For example:

```
#pragma link C++ class MyClass*;
```

This will request the dictionary for all the class whose name start with `'MyClass'` and are already known to Cling (class templates need to have already been instantiated to be considered).

```
#pragma link [C|C++|off] all [class|function|global|typedef];
```

This pragma statement turns on or off the dictionary generation for all classes, structures, namespaces, global variables, global functions and typedefs seen so far by Cling. Example:

```
// some C++ header definition
#ifdef __ROOTCLING__
// turns off dictionary generation for all
#pragma link off all class;
#pragma link off all function;
#pragma link off all global;
#pragma link off all typedef;
#endif
```

The next pragma statement selectively turns on or off the dictionary generation for the specified `classs`, `struct`, `union`, `enum` or `namespace`:

```
#pragma link
[C|C++|off][class|class+protected|
struct|union|enum|namespace][name];
```

The Dictionary of all public members of class and struct will be generated. If the '`class+protected`' flag is used, the dictionary for protected members will also be generated. However, dictionary for protected constructor and destructor will not be generated. This '`class+protected`' flag will help you only for plain protected member access, but not for virtual function resolution.

If you use the '`namespace`' flag, it is recommended to add also:

```
#pragma link C++ nestedclass;
#pragma link C++ nestedtypedef;
```

The behavior of '`class`', '`struct`' and '`namespace`' flag are identical. Example:

```
// some C++ header definition
#ifdef __ROOTCLING__
#pragma link off all class;
#pragma link C++ class A;
#pragma link C++ class B;
#pragma link C++ class C<int>;
#pragma link C++ class+protected D;
#pragma link C++ namespace project1;
#pragma link C++ nestedclass;
#pragma link C++ nestedtypedef;
#endif
```

The next pragma statement selectively turns on or off the dictionary generation for global variables and typedef.

```
#pragma link [C|C++|off] [global|typedef] [name];
```

Example:

```
// some C/C++ header definition
#ifdef __ROOTCLING__
#pragma link off all global;
#pragma link off all typedef;
#pragma link C++ global a;
#pragma link C++ typedef Int_t;
#endif
```

This pragma statement turns on the dictionary generation for nested classes and nested typedefs.

```
#pragma link [C|C++|off] [nestedclass|nestedtypedef];
```

Example:

```
// some C/C++ header definition
#ifdef __ROOTCLING__
#pragma link off all global;
#pragma link off all typedef;
#pragma link C++ global a;
#pragma link C++ typedef Int_t;
#endif
```

The next pragma statements turn on or off the dictionary generation for the specified function(s) or member function(s). The list of arguments' type is optional. If you omit argument types, all function with specified [`name`] will be affected. If the list of arguments' type is specified, only the function that has exactly same argument list will be affected.

```
#pragma link [C++|C|off|MACRO] function [fname]<(argtypes)>;
#pragma link
[C++|C|off|MACRO] function [classname]::[fname]<(argtypes)>;
```

The '`#pragma link [C++|C] function`' and '`#pragma link MACRO function`' behaves similarly. The '`#pragma link [C++|C] function`' assumes the target to be a real function which has pointer to it. A pointer to registered function is registered. On the other hand, '`#pragma link MACRO function`' assumes target to be macro function. Pointer to function cannot be referenced in this case.

For the next example:

```
void f(int a);
void f(double a);
int g(int a,double b);
int g(double x);
#define max(a,b) (a>b?a:b)

class A {
   public:
      int h(double y);
      int h(int a,double b);
};
```

The pragma statements are:

```
#ifdef __ROOTCLING__
#pragma link off all functions;
#pragma link C++ function f;
#pragma link C++ function g(int,double);
#pragma link C++ MACRO max;
#pragma link C++ class A;
#pragma link off function A::h(double);
#endif
```

Until Cling version 5.15.60, in order to generate dictionary for a member function, not only the member function but also the class itself has to be turned on for the linkage. There was an inconvenience when generating dictionary for template member function afterwards.

From Cling v.5.15.61, a new behavior is introduced. If link for a member function is specified, dictionary is generated even if link to the belonging class is off. For example, if you originally have A.h as follows:

```
// A.h
template<class T> class A {
   public:
      template<class E> void f(E& x) { ... }
};
```

And generate dictionary for that:

```
#ifdef __ROOTCLING__
#pragma link C++ class A<int>;
#endif
```

Then prepare another header file and instantiate the template member function of A.:

```
// B.h
#include "A.h"

class B {
...
};
```

You can generate dictionary for the newly instantiated template member function only.

```
#ifdef __ROOTCLING__
#pragma link off defined_in A.h;
#pragma link C++ function A<int>::f(B&);
#endif
```

The next pragma turns off the dictionary generation of all the member functions in all classes.

```
#pragma link off all methods;
```

Example:

```
#ifdef __ROOTCLING__
#pragma link off all methods;
#endif
```

The next pragma statements control the linking of all the member functions or data members for a specified class.

```
#pragma link [C|C++|off] all_function   [classname];
#pragma link [C|C++|off] all_datamember [classname];
```

At this moment, there should be no needs to use those statements. Example:

```
#ifdef __ROOTCLING__
#pragma link off all_function   A;
#pragma link off all_datamember A;
#endif
```

See also: `#pragma link function`.

The next pragma statement turns on/off dictionary generation of the object defined in specific file. The filename has to be the full pathname of the file.

```
#pragma link [C|C++|off] defined_in [filename];
```

Example:

```
// file1.h
// any C++ header definition

// file2.h

#ifdef __ROOTCLING__
#pragma link off all classes;
#pragma link off all functions;
#pragma link off all globals;

#pragma link off all typedef;
#pragma link C++ defined_in file1.h;
#endif
```

The next pragma statements turn on or off the dictionary generation of the object defined in a specific scope. The [`scope_name`] should be `class` name, `struct` name or `namespace` name. When using these pragmas, it is recommended to use also:

```
#pragma link C++ nestedclass
```

Otherwise, definitions in enclosed scope do not appear in the dictionary.

```
#pragma link [C|C++|off] defined_in [scope_name];
#pragma link [C|C++|off] defined_in
[class|struct|namespace] [scope_name];
```

Example:

```
namespace ns {
   int a;
   double b;
};
```

The pragma statements are:

```
#ifdef __ROOTCLING__
#pragma link C++ defined_in ns;
#pragma link C++ nestedclass;
#endif
```

This statements controls default link mode for `rootcling`.

```
#pragma link default [on|off]
```

By turning default 'on', all language constructs in given header files will be included in generated Cling dictionary (interface method source file). If default is set to 'off', nothing will be included in the generated dictionary. The next statement explicitly set linkage to each item:

```
#pragma link [C|C++|off] [class|function|global]
```

This pragma statement must be given before `rootcling` reads any C/C++ definitions from header files. Example:

```
#ifdef __ROOTCLING__
#pragma link default off;
```

```
#endif

class A {
   int a;
   double b;
};

class B {
   int d;
   double e;
};

#ifdef __ROOTCLING__
#pragma link C++ class A;      // only class A is linked, not B
#endif
```

1.5.1.2.1 Compilation

Step 4: Compile the class using the `Makefile`.In the `Makefile` call `rootcling` to make the dictionary for the class. Call it `SClassDict.cxx`. The `rootcling` utility generates the methods `Streamer`, `TBuffer` &operator>>() and `ShowMembers`for ROOT classes.

```
gmake -f Makefile
```

Load the shared library:

```
root[] .L SClass.so
root[] SClass *sc = new SClass()
root[] TFile *f = new TFile("Afile.root","UPDATE");
root[] sc->Write();
```

For more information on `rootcling` see the `$ROOTSYS/test` directory `Makefile`, `Event.cxx`, and `Event.h` for an example, or follow this link: http://root.cern.ch/root/RootCintMan.html

1.6 genreflex: A Comfortable Interface to rootcling

Version 5 supported both `Cint` and `Reflex` dictionaries. The tool to create `Reflex` dictionaries was a Python script called `genreflex` and was very successful in the user community. Even if version 6 has only one type of dictionaries, `cling` dictionaries, a re-implementation of `genreflex` is provided. More precisely, in ROOT6, `genreflex` is nothing but a wrapper around `rootcling`, which offers an identical CLI and behaviour to the old Python tool. The input to `genreflex` is a C++ header file, a set of switches and a *selection XML file*. The output, as for `rootcling`, is a C++ dictionary source and a `pcm` files. An exhaustive documentation of the CLI switches of `genreflex` can be inspected with the `genreflex --help` command.

The entity corresponding to the `LinkDef` file for `genreflex` is the *selection XML file*, also called *selection XML* or simply *selection file*. A *selection XML file* allows to describe a list of classes for which the dictionaries are to be created. In addition, it allows to specify properties of classes or data members, without the need to add comments in the source code. This is of primary importance when dictionaries must be created for classes residing in code which cannot be modified. For a complete description of the structure of the *selection XML files* and the way in which attributes can be set, refer to the `genreflex --help` command.

It is important to observe that *selection XML files* can be used in presence of `rootcling` invocations instead of `LinkDef` files.

1.6.1 The `ROOT::Meta::Selection` namespace

Not only `LinkDef` and `selection` files allow to select the classes for which the dictionaries must be created: a third method is available. This is represented by the `ROOT::Meta::Selection` namespace. The idea behind this technique is that all the classes which are located in this special namespace are automatically selected for dictionary generation. All the properties and annotations allowed by `LinkDef` and `selection XML` files are possible. For a detailed documentation of the features of the `ROOT::Meta::Selection` namespace, refer to its online documentation.

1.7 Adding a Class with ACLiC

Step 1: Define your class

```
#include "TObject.h"

// define the ABC class and make it inherit from TObject so that
// we can write ABC to a ROOT file
class ABC : public TObject {

   public:
      Float_t a, b, c, p;
      ABC() : a(0), b(0), c(0), p(0){};

// Define the class for the dictionary
   ClassDef (ABC,1)
};

// Call the ClassImp macro to give the ABC class RTTI and
// full I/O capabilities.

#if !defined(__CLING__)
ClassImp(ABC);
#endif
```

Step 2: Load the ABC class in the script.

```
   // Check if ABC is already loaded
   if (!TClass::GetDict("ABC")) {
      gROOT->ProcessLine(".L ABCClass.C++");
   }

   // Use the Class
   ABC *v = new ABC;
   v->p = (sqrt((v->a * v->a)+ (v->b * v->b)+(v->c * v->c)));
```