A short, pragmatic and INCOMPLETE intro to C++/ROOT programming.

Mikołaj Krzewicki
Nikhef/UU

C++

- Middle-level, statically typed, free-form, multi-paradigm, compiled language(wikipedia.org).
- Development started in 1979 by Bjarne Stroustrup at Bell Labs (C with classes)
  - Renamed to “C++” in 1983
- One of most popular programming languages ever created.

C++ middle level language

- High level abstractions:
  - Hardware 'independent' data types (int, float,...)
  - Data structures (classes, arrays, streams,...)
  - Execution flow control (functions, methods, loops,...)
  - Portability (to a reasonable degree)
  - Additionally a large standard (and non-standard) library
- Low level features:
  - Explicit memory access & management (!)
  - Bit-level operations
  - ...

C++ multi paradigm language

- Procedural programming:
  - Use of subroutines (functions) to add structure to the program
- Object-oriented programming:
  - Association of data structures and functions (methods) that can operate on them
- ...

C++/ROOT

- ROOT is a C++ framework and environment used and developed at CERN.
- Gives you ready to use data types (histograms, trees) and constructs for math, IO and more...
- Provides an interpreter (no need for compilation)
- No need to write independent programs.
- All in all simplifies a lot of tasks (most of the time).
- Full documentation at http://root.cern.ch/

Programming (in C/C++/ROOT/...)

- Declare what data we want to operate on (memory)
- Make functions to operate on the data (cpu)

Declaring variables

- Reserves space in memory to hold data of specific type (integer, float, structure,pointer)
- The curly brackets { } define scope – a logical block with it's own local data (and instructions)
- Inside a block only data explicitly declared in there are available (for now)
- Closing brace deletes all declared variables from memory

```
{
  int i=0;
  float f=3.1415;
}
```

Declaring functions

- Functions are named blocks that can return a value of a certain type.
- Once defined a function can be executed (called) anywhere (almost).

```
int MyFunction()
{
  int i=0;
  return i;
}

int DoSomething()
{
  int result;
  result=MyFunction();
  return result;
}
```

Declaring functions

- Functions can take input of a certain type.
  Just like mathematical equivalents:
  `y=f(x1,x2,..)`
- The (call to) function can be used as a value of its return type – in this case an integer (it doesn't matter if you use y or `f(x)` in an expression)

```
int MyFunction(int input)
{
  int i=0;
  i+=input;
  return i;
}


int DoSomething()
{
  int result=0;
  int value=2;
  result=MyFunction(value);
  return result;
}
```

Essential basics

- Checking for a condition:

```
if (condition) {…}
else {…}
```

condition is either (something that returns a) true or false or NULL or not NULL for pointers (you can check if a pointer points to something useful)

More essential basics

- Most arithmetic operators work 'intuitively', e.g.
  - `a=b, a+b, a-b, a*b, a/b, a%b`
- Beware of implicit type conversions: results depend on type of arguments: 3.0/2=1.5 but 3/2=1 (integer division)
- Comparison operators, result is always a boolean:
  - a==b, a<b, a<=b
  - Beware of comparison a==b vs assignment a=b.

Loops

- Often you want to repeat some calculation a specified number of times, or repeat it until some condition occurs
- Most commonly used: for-loop.
- Syntax:

  for(init;condition;finish)`{...}`

  init executed once after `{`

  condition (must be `bool`) every iteration after `{`

  finish at `}` (also every time)

```
int MyFunction(int input)
{
  return ++input;
}

int DoSomething()
{
  int value;
  for (int i=0; i<20; i++)
  {
    value=MyFunction(value);
  }
  return value;
}
```

Loops

- Controlling loops:

  `continue` – skips to the next iteration

  `break` – jumps out of the loop, no more iterations

```
for (Int_t i=0; i<1000 ;i++)
{
  if (i>10) {break;} //after 10 no more loop
  if (i>0) {continue;}
  printf("this text will appear only once\n");
}
```

Final touches: prototypes and #include statement

- A compiler (or interpreter) runs over the code from top to bottom while compiling/parsing it.
- Sometimes it is not desirable/possible to always have the functions implemented before you use them.
- External libraries: we don't have to reinvent the wheel, we can use someone elses code.

```
int main(){
 printf(“result is: %i\n”,DoSomething());
}

int MyFunction(int input){
  return ++input;
}

int DoSomething(){
  int value;
  for (int i=0; i<20; i++){
    value=MyFunction(value);
  }
 return value;
}
```

Final touches: prototypes and #include statement

- Placing prototypes of functions at the beginning makes them available throughout the code.
- If the prototypes are specified in a separate file (as is a case with external libraries) we can #include it with the same benefit.
- Of course the implementation has to be there somewhere (in the search path of the linker) or else the compiler will complain.

```
#include <cstdio>
int MyFunction(int input);
int DoSomething();

int main(){
  printf(“result is: %i\n”,DoSomething());
}

int MyFunction(int input){
  return ++input;
}

int DoSomething(){
  int value;
  for (int i=0; i<20; i++){
    value=MyFunction(value);
  }
 return value;
}
```

Exercise: compile&run

- Type the code and save in a file named test.cxx; compile it with:

  `g++ test.cxx -o test`

- It will produce an executable file test which you can run in the terminal a few times by typing:

  ./test

- Do you understand the output?

```
#include <cstdio>
int MyFunction(int input);
int DoSomething();

int main(){
  printf(“result is: %i\n”,DoSomething());
}

int MyFunction(int input){
  return ++input;
}

int DoSomething(){
  int value;
  for (int i=0; i<20; i++){
    if (i=10) continue;
    value=MyFunction(value);
  }
 return value;
}
```

Object-oriented programming with ROOT

- An object is a data structure
- Can contain simple data (integers, floats,...) as well as other objects (members)
- Objects have a type: their class
- The class also specifies the operations that are allowed for manipulation of the data it holds, most often the data members are not directly accessible (only by methods).
- members/methods only accessible if made public.
- Much much more (like inheritance, friends, ...) will not be covered now.

```
class ExampleHistogramType
{
  protected:
    …
    Int_t fNentries;
    Int_t fNbins;
    …
  public:
    …
    Int_t GetEntries();
    Int_t GetNbins();
    …
}
```

Objects

- Instantiation of an object goes like with any other variable
- Special syntax for accessing members and methods:
  object.member
  object.method()

```
#include <TH1F.h>
#include <TRandom.h>
void PlotHistogram()
{
  TH1F histogram;
  TRandom rndGener;
  histogram.SetBins(50,-4.0,4.0);
  for (Int_t i=0;i<10000;i++)
  {
    Float_t rndNumber;
    rndNumber=rndGener.Gaus(0.0,1.0);
    histogram.Fill(rndNumber);
  }
  histogram.DrawCopy();
}
```

Exercise: objects

- Call the file PlotHistogram.C
- Start root by typing root in the terminal
- Run the code: .x PlotHistogram.C
- Change last line to: histogram.Draw() and run

- What happened?

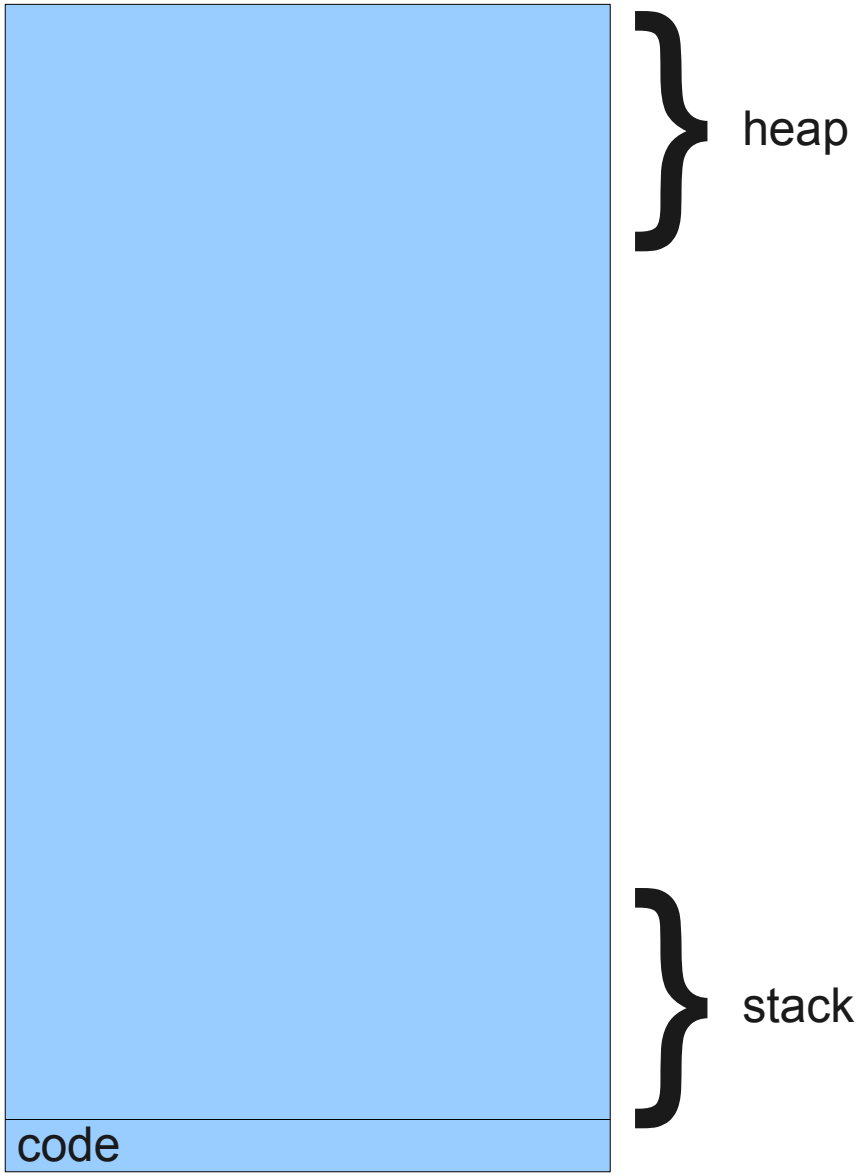
```
#include <TH1F.h>
#include <TRandom.h>
void PlotHistogram()
{
  TH1F histogram;
  TRandom rndGener;
  histogram.SetBins(50,-4.0,4.0);
  for (Int_t i=0;i<10000;i++)
  {
    Float_t rndNumber;
    rndNumber=rndGener.Gaus(0.0,1.0);
    histogram.Fill(rndNumber);
  }
  histogram.DrawCopy();
}
```

Dynamic memory, pointers, references

- Sometimes it is useful to have the data outlive the scope boundaries
- Sometimes we don't know a priori how much memory to reserve (eg. for an array)
- local variables are allocated on the stack and destroyed when they go out of scope.
- We can also allocate memory in a different memory pool (the heap) using the “new” keyword – this is persistent, so we have to make sure we also free the allocated memory after we're done with it

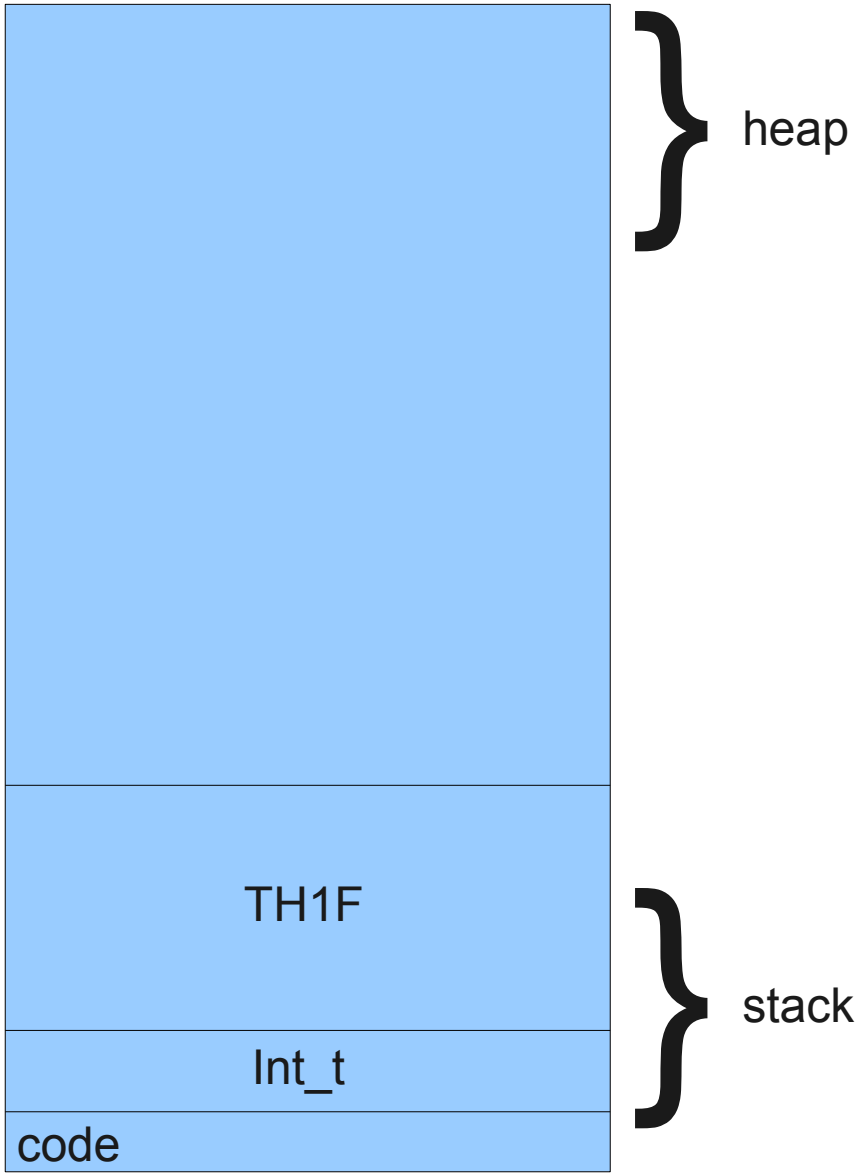
Dynamic memory

```
#include <TH1F.h>


void PlotHistogram()
{
  {
    Int_t i;
    TH1F histogram;
    new TH1F;
  }
}
```



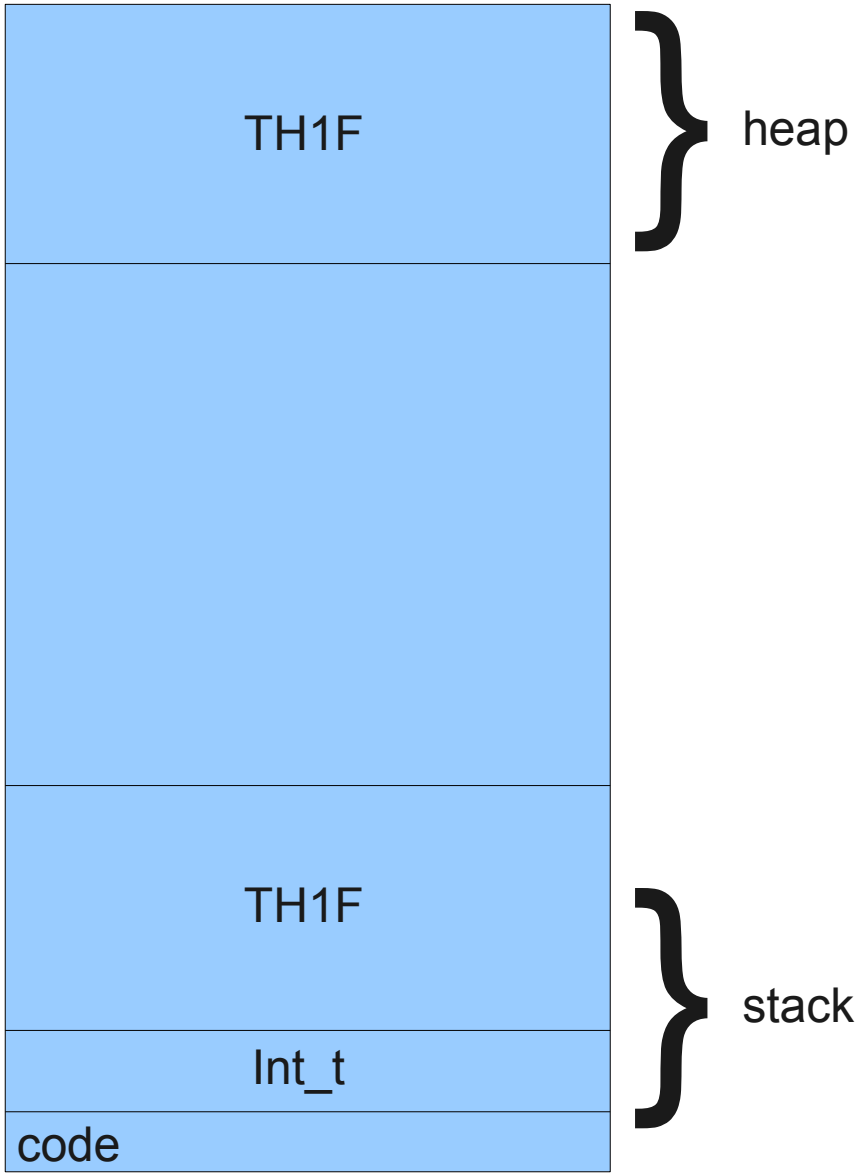
Dynamic memory

```
#include <TH1F.h>

void PlotHistogram()
{
  {
    Int_t i;
    TH1F histogram;
    new TH1F;
  }
}
```



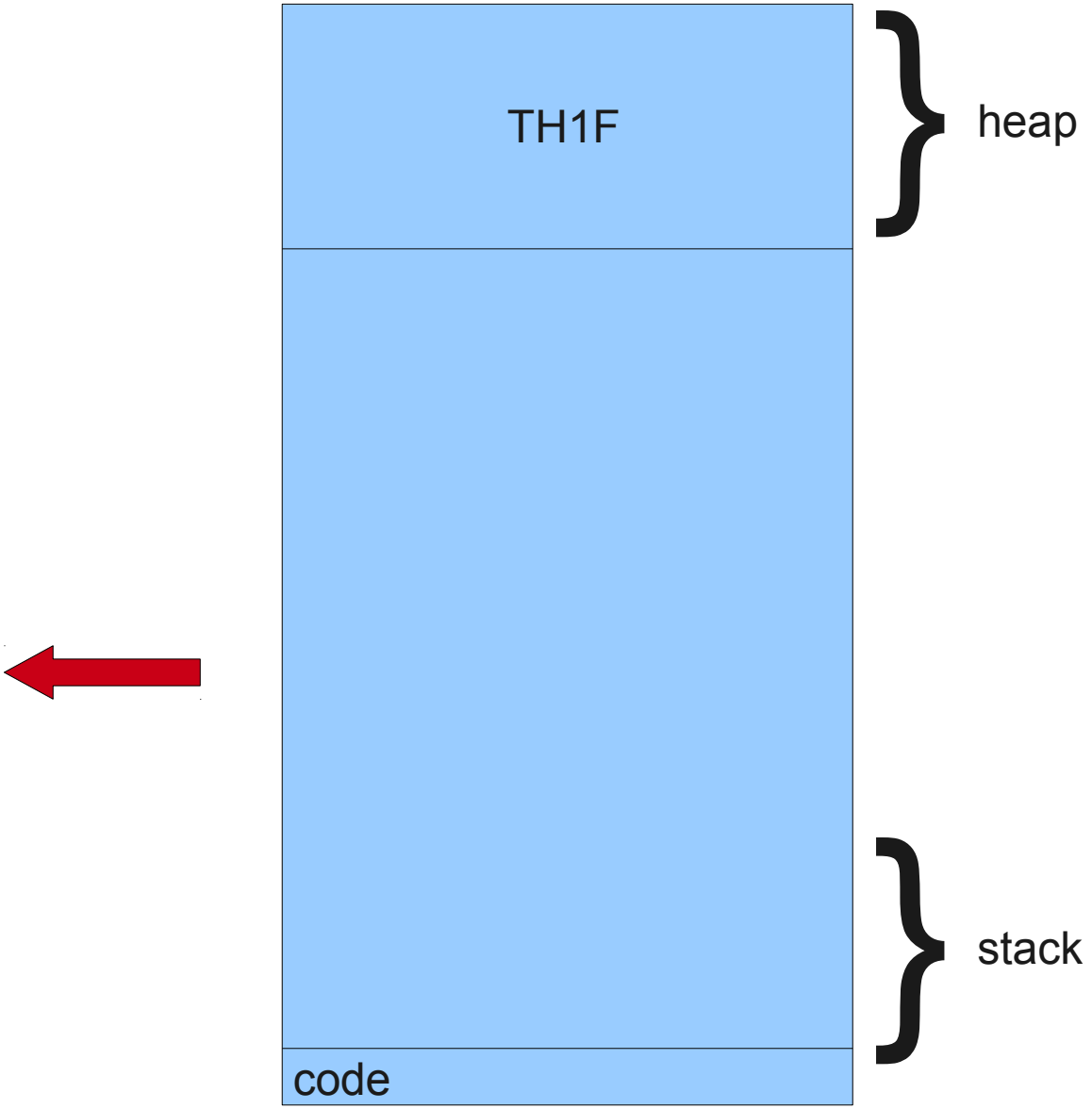
Dynamic memory

```
#include <TH1F.h>

void PlotHistogram()
{
  {
    Int_t i;
    TH1F histogram;
    new TH1F;
  }
}
```



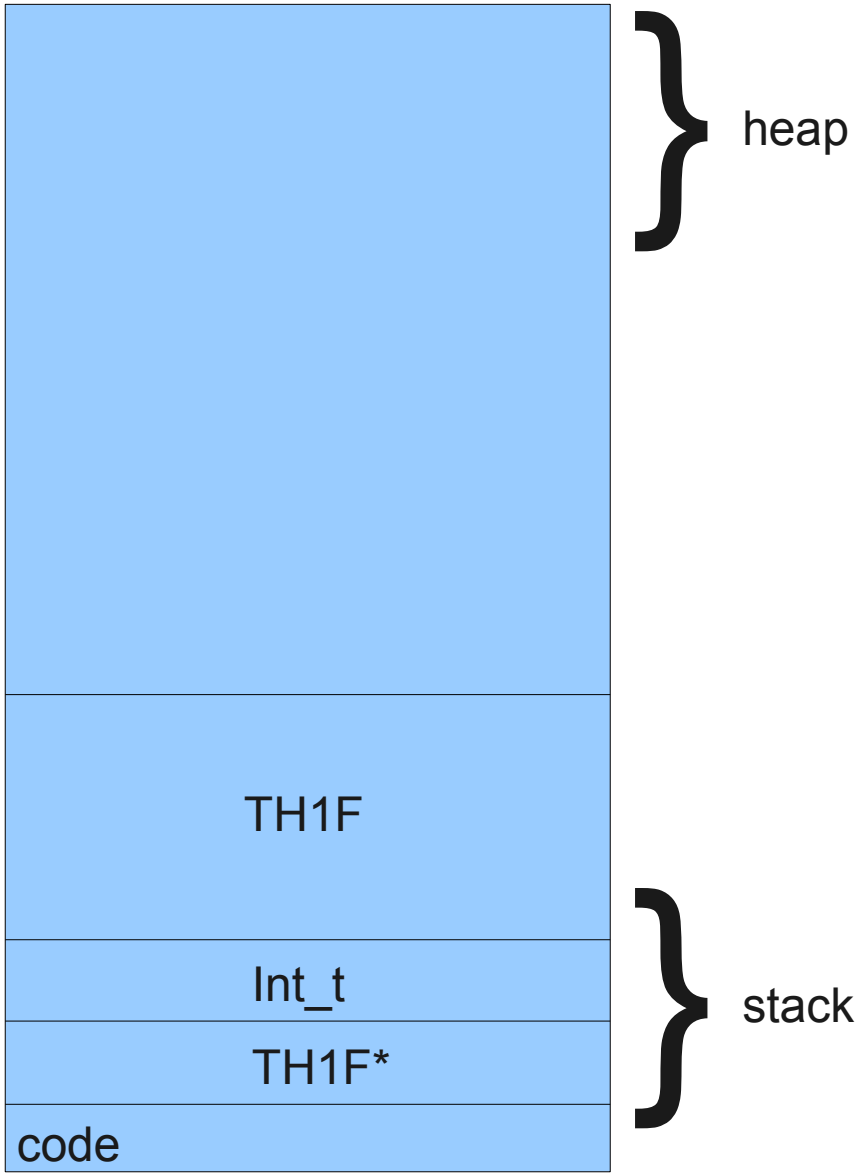
Dynamic memory

```
#include <TH1F.h>

void PlotHistogram()
{
  {
    Int_t i;
    TH1F histogram;
    new TH1F;
  }
  //object still there
  //but no way to get to it
}
```



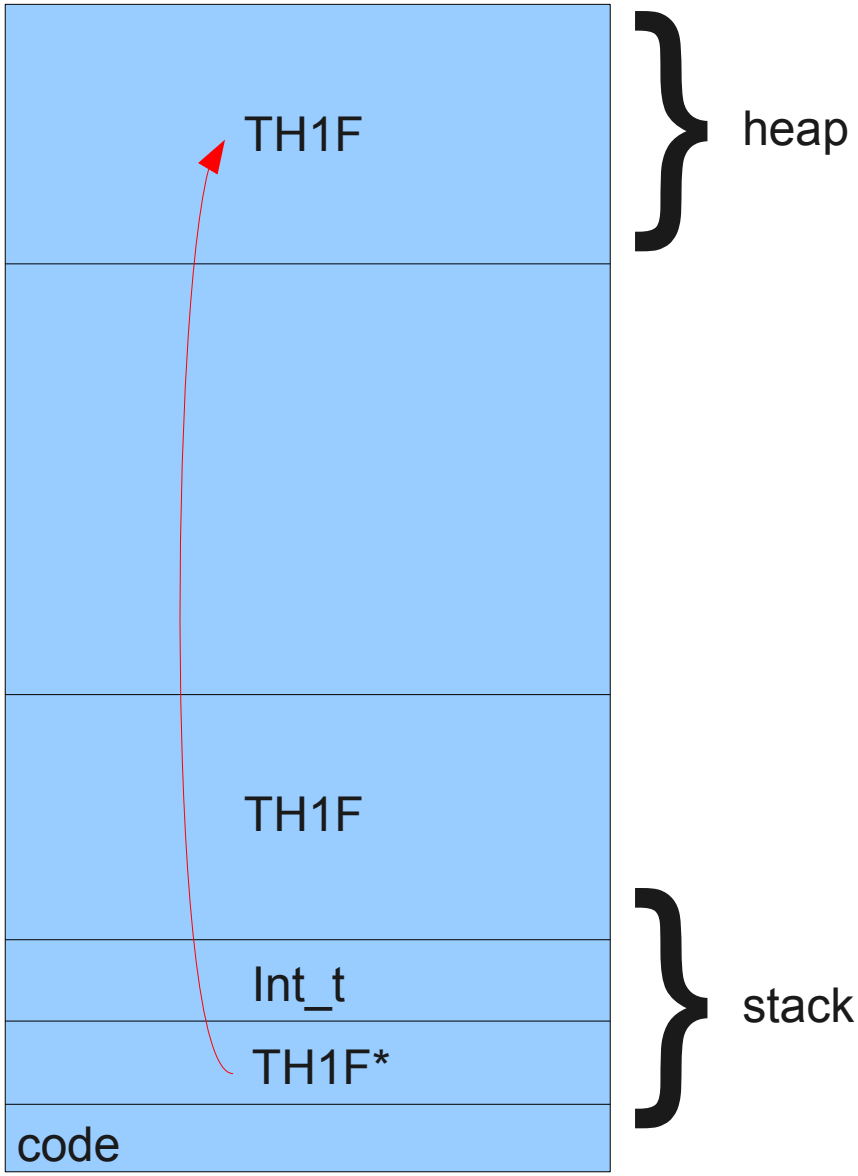
Pointers

```
#include <TH1F.h>

void PlotHistogram()
{
  //pointer type
  //can only contain an address
  //of an object or NULL
  TH1F* p_histogram=NULL;
  {
    Int_t i;
    TH1F histogram;
    p_histogram = new TH1F;
  }
}
```



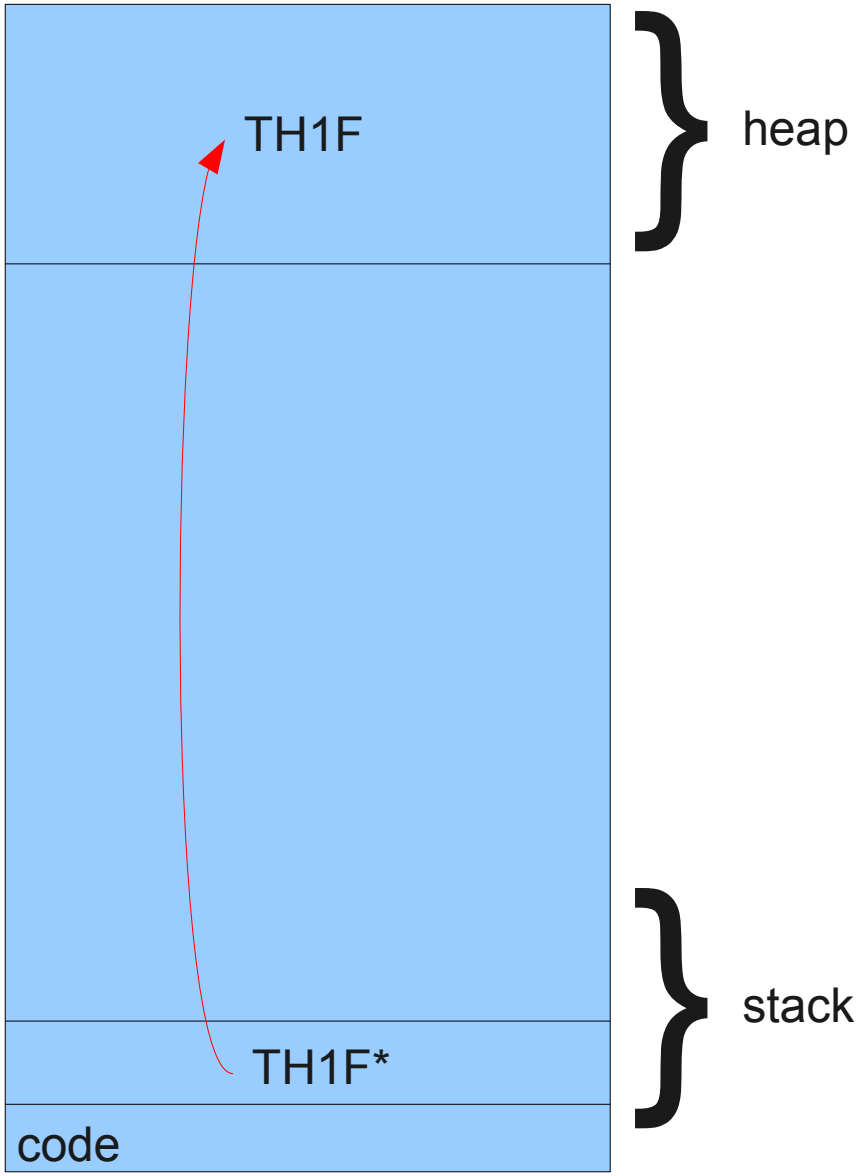
Pointers

```
#include <TH1F.h>

void PlotHistogram()
{
  //pointer type
  //can only contain an address
  //of an object or NULL
  TH1F* p_histogram=NULL;
  {
    Int_t i;
    TH1F histogram;
    p_histogram = new TH1F;
  }
}
```



Pointers

```
#include <TH1F.h>

void PlotHistogram()
{
  //pointer type
  //can only contain an address
  //of an object or NULL
  TH1F* p_histogram=NULL;
  {
    Int_t i;
    TH1F histogram;
    p_histogram = new TH1F;
  }
  delete p_histogram;
}
```



Pointers & addresses by example

- Syntax:

```
TH1F* pointerToHistogram;                // declaration, contains address

TH1F histogram;                          // ”regular” object on stack
pointerToHistogram = &histogram;         // now it points to histogram

TH1F histogram2;
histogram2 = histogram1;                 // possible, both are regular objects
histogram2 = *pointerToHistogram;        // otherwise the types clash

// so in short: & gets the address of an object
//              * returns the actual object referenced by the pointer

// also note:
// &pointerToHistogram will get us the address of the pointer(address of address)
```

Pointers & objects by example

```
//let's make an object to point to:
TH1F* pointerToHistogram = new TH1F;

//the following will NOT work:
pointerToHistogram.Draw();     //pointers have no methods

//this WILL work:
(*pointerToHistogram).Draw();  //call to method of object

//mostly used with this shorthand:
pointerToObject->Draw();
```

Some ROOT types

- Some more commonly used ROOT types:

  `Bool_t` – boolean (kTRUE,kFALSE)

  `Int_t` – integer

  `Float_t` – floating point number

  `Double_t` – double-precision float

How to get to out software

- Boot Linux
- Login
- Open a terminal
- Type:

  ```
  . /home/projects/ns-sap/ns-sap-env.sh v4-19-Rev-05
  ```
- (there is a dot-space at the beginning)
- You can start root by typing:

  ```
  root
  ```

Example

- Start root.
- Run this program in root:

  `.x PlotHistogram.C`

- Do you understand what the program does?
- Note also that we no longer need to use:

  `TH1F::DrawCopy()`

  why?

```
#include <TH1F.h>
#include <TRandom.h>
void PlotHistogram()
{
  TH1F* histogram=new TH1F;
  histogram->SetBins(50,-4.0,4.0);
  TRandom rndGener;
  for (Int_t i=0;i<10000;i++)
  {
    Float_t rndNumber;
    rndNumber=rndGener.Gaus(0.0,1.0);
    if (i=10) continue;
    if (rndNumber>0)
      histogram->Fill();
  }
  histogram->Draw();
}
```