Scientific Computing WS 2018/2019

Lecture 3

Jürgen Fuhrmann

juergen.fuhrmann@wias-berlin.de

Recap from last time

C++: first steps

Standard features:

- scalar types
- basic operations
- flow control
- $\Rightarrow$ see e.g. online references `http://www.cplusplus.com/`, `https://en.cppreference.com/w`

The Preprocessor

- Before being sent to the compiler, the source code is sent through the *preprocessor*
- It is a legacy from C which is slowly being squeezed out of C++
- Preprocessor commands start with #
- Include contents of file `file.h` found on a default search path known to the compiler:

```
#include <file.h>
```

- Include contents of file `file.h` found on user defined search path:

```
#include "file.h"
```

- Define a piece of text (mostly used for constants in pre-C++ times) (avoid, use `const` instead):

```
#define N 15
```

- Define preprocessor macro for inlining code (avoid, use inline functions instead):

```
#define  MAX(X,Y) (((x)>(y))?(x):(y))
```

Conditional compilation and pragmas

- Conditional compilation of pieces of source code, mostly used to dispatch between system dependent variant of code. Rarely necessary nowadays...

```
#ifdef MACOSX
statements to be compiled only for MACOSX
#else
statements for all other systems
#endif
```

- There can be more complex logic involving constant expressions
- A pragma gives directions to the compiler concerning code generation:

```
#pragma omp parallel
```

Headers

- If we want to use functions from the standard library we need to include a *header file* which contains their declarations
  - The `#include` statement invokes the C-Preprocessor and leads to the inclusion of the file referenced therein into the actual source
  - Include files with names in < > brackets are searched for in system dependent directories known to the compiler

```
#include <iostream>
```

Namespaces

- Namespaces allow to prevent clashes between names of functions from different projects
  - All functions from the standard library belong to the namespace `std`

```
namespace foo
{
  void cool_function(void);
}

namespace bar
{
  void cool_function(void);
}

...

{
  using namespace bar;
  foo::cool function()
  cool_function() // equivalent to bar::cool_function()
}
```

Modules ?

- Currently, C++ has no well defined module system.
- A module system usually is emulated using the preprocessor and namespaces.

Emulating modules

- File `mymodule.h` containing interface declarations

```
#ifndef MYMODULE_H // Handle multiple #include statements
#define MYMODULE_H
namespace mymodule
{
  void my_function(int i, double x);
}
#endif
```

- File `mymodule.cpp` containing function definitions

```
#include "mymodule.h"
namespace mymodule
{
  void my_function(int i, double x)
  {
    ...body of function definition...
  }
}
#endif
```

- File using `mymodule`:

```
#include "mymodule.h"
...
mymodule::my_function(3,15.0);
```

main

Now we are able to write a complete program in C++

- `main()`
  is the function called by the system when running the program. Everything else needs to be called from there.

Assume the follwing content of the file `run42.cxx`:

```
#include <cstdio>

int main()
{
   int i=4,j=2;
   int answer=10*i+j;
   printf("Hello world, the answer is %d!\n",answer);
   return 0;
}
```

Then the sequence of command line commands

```
$ g++ -o run42 run42.cxx
$ ./run42
```

gives the right answer to (almost) anything.

Command line instructions to control compiler

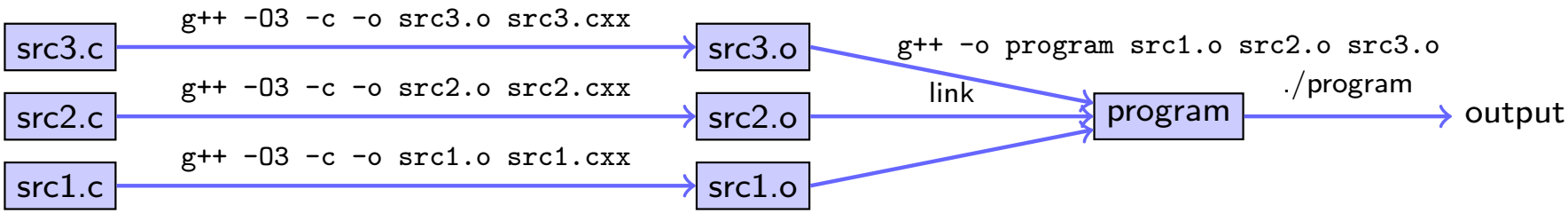
- By default, the compiler command performs the linking process as well
- Compiler command (Linux)

```
g++       GNU C++ compiler
g++-5     GNU C++ 5.x
clang++   CLANG compiler from LLVM project
icpc      Intel compiler
```

- Options (common to all of those named above, but not standardized)

```
-o name              Name of output file
-g                   Generate debugging instructions
-O0, -O1, -O2, -O3   Optimization levels
-c                   Avoid linking
-I<path>             Add <path> to include search path
-D<symbol>           Define preprocessor symbol
-std=c++11           Use C++11 standard
```

Compiling. . .




```
$ g++ -O3 -c -o src3.o src3.cxx
$ g++ -O3 -c -o src2.o src2.cxx
$ g++ -O3 -c -o src1.o src1.cxx
$ g++ -o program src1.o src2.o src3.o
$ ./program
```

Shortcut: invoke compiler and linker at once

```
$ g++ -O3 -o program src1.cxx src2.cxx src3.cxx
$ ./program
```

Some shell commands in the terminal window

```
ls -l              list files in directory
                   subdirectories are marked with 'd'
                   in the first column of permission list
cd  dir            change directory to dir
cd  ..             change directory one level up in directory hierachy
cp  file1 file2    copy file1 to file2
cp  file1 dir      copy file1 to directory
mv  file1 file2    rename file1 to file2
mv  file1 dir      move file1 to directory
rm  file           delete file
[cmd] *.o          perform command on all files with name ending with .o
```

Editors & IDEs

- Source code is written with text editors
  (as compared to word processors like `MS Word` or `libreoffice`)
- Editors installed are
  - `gedit` - text editor of gnome desktop (recommended)
  - `emacs` - comprehensive, powerful, a bit unusual GUI (my preferred choice)
  - `nedit` - quick and simple
  - `vi`, `vim` - the UNIX purist's crowbar
    (which I avoid as much as possible)
- Integrated development environments (IDE)
  - Integrated editor/debugger/compiler
  - eclipse (need to get myself used to it before teaching)

Working with source code

- Copy the code:

```
$ cp /net/wir/examples/part1/example.cxx .
```

- Editing:

```
$ gedit  example.cxx
```

- Compiling: (-o gives the name of the output file)

```
$ g++ -std=c++11  example.cxx -o  example
```

- Running: (./ means file from current directory)

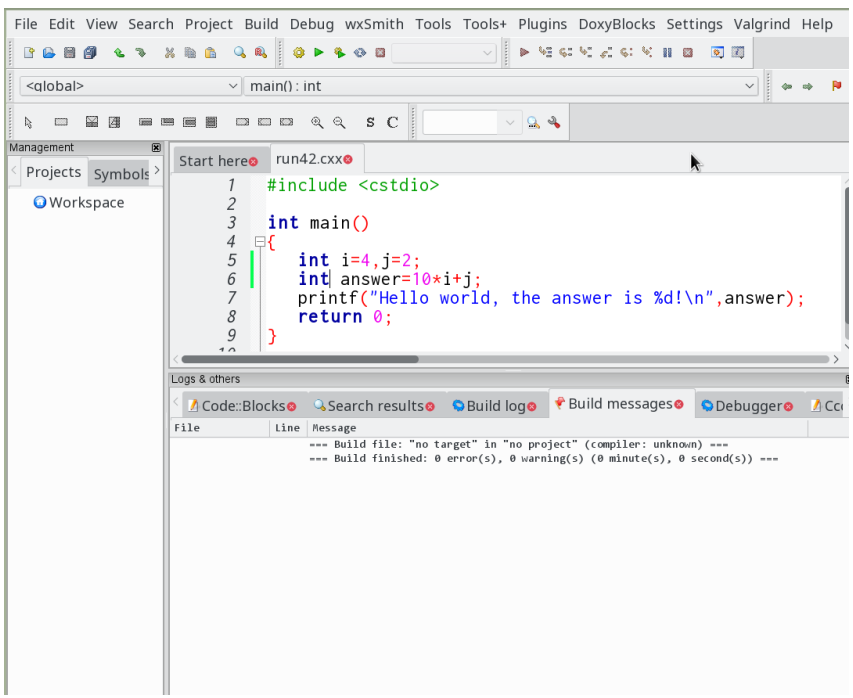
```
$ ./example
```

Alternative:Code::Blocks IDE

- `http://www.codeblocks.org/`
- Open example

```
$ codeblocks example.cxx
```

- Compile + run example: `Build/"Build and Run"` or `F9`
- Switching on C++11 standard: tick `Settings/Compiler/"Have g++ follow the C++11..."`



Addresses and pointers

- Objects are stored in memory, in order to find them they have an *address*
- We can determine the address of an object by the `&` operator
  - The result of this operation can be assigned to a variable called *pointer*
  - "pointer to type `x`" is another type denoted by `*x`
- Given an address (pointer) object we can refer to the content using the `*` operator

```
int i=15;  // i is an object
int *j= &i; // j is a pointer
int k=*j; // now, k=15
```

- The `nullptr` object can be assigned to a pointer in order to indicate that it points to "nothing"

```
int *p=nullptr;
```

Passing addresses to functions

- Instead of values, addresses can be passed to functions

```
void do_multiplication(double x, double y, double *result)
{
    *result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,&result) // result now contains 45
```

Arrays

- Focusing on numerical methods for PDEs results in work with finite dimensional vectors which are represented as *arrays* - sequences of consecutively stored objects
- Stemming from C, in C++ array objects represent just the fixed amount of consecutive memory. No size info or whatsoever
- No bounds check
- First array index is always 0

```
double x[9]; // uninitialized array of 9 elements
double y[3]={1,2,3}; // initialized array of 3 elements
double z[]={1,2,3};  // Same
double z[]{1,2,3};   //Same
```

- Accessing arrays
  - [] is the array access operator in C++
  - Each element of an array has an index

```
double a=x[3];  // undefined value because x was not initialized
double b=y[12]; // undefined value because out of bounds
y[12]=19;       // may crash program ("segmentation fault"),
double c=z[0];  // Acces to first element in array, now c=1;
```

Arrays, pointers and pointer arithmetic

- Arrays are strongly linked to pointers
- Array object can be treated as pointer

```
double x[]={1,2,3,4};
double b=*x; // now x=1;
double *y=x+2; // y is a pointer to third value in arrax
double c=*y;   // now c=3
ptrdiff_t d=y-x; // We can also do differences between pointers
```

- Pointer arithmetic is valid only in memory regions belonging to the same array

Arrays and functions

- Arrays are passed by passing the pointer referring to its first element
- As they contain no length information, we need to pass that as well

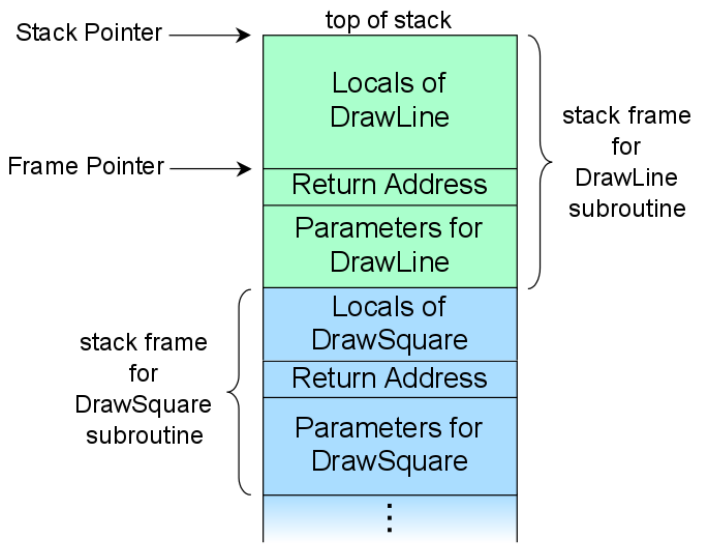
```
void func_on_array1(double[] x, int len);
void func_on_array2(double* x, int len); // same
void func_on_array3(const double[] x, int len);//same, but prevent changing x
...
double x[]={3,4,5};
int len=sizeof(x)/sizeof(x[0]);
func_on_array1(x,len);
```

- Be careful with array return

```
double * some_func(void)
{
  double a[]={-1,-2,-3};
  return a; // illegal as with the end of scope, the life time of a is over
            // smart compilers at least warn
}
```

Memory: stack

- pre-allocated memory where `main()` and all functions called from there put their data.
  - Any time a function is called, the current position in the instruction stream is stored in the stack as the return address, and the called function is allowed to work with the memory space after that




```
void DrawLine(double x0, double y0, double x1, double y1)
{
  paint ...
}

void DrawSquare(double x0,double y0, double a)
{
  DrawLine(x0,y0,x0+a,y0);
  DrawLine(x0+a,y0,x0+a,y0+a);
  DrawLine(x0+a,y0+a,x0,y0+a);
  DrawLine(x0,y0+a,x0,y0);
}
```

By R. S. Shaw, Public Domain,

https://commons.wikimedia.org/w/index.php?curid=1956587

Stack space is scarce

- Variables declared in `{}` blocks get placed on the stack
- All previous examples had their data on the stack, even large arrays
- Stack space should be considered scarce
- Stack size for a program is fixed, set by the system
- On UNIX, use `ulimit -s` to check/set stack size default

Memory: heap

- Chunks from free system memory can be reserved – "allocated" – on demand in order to provide memory space for objects
- The operator `new` reserves the memory and returns an address which can be assigned to a pointer variable
- The operator `delete` (`delete[]` for arrays) releases this memory and makes it available for other processes
- Compared to declarations on the stack, these operations are expensive
- Use cases:
  - Problem sizes unknown at compile time
  - Large amounts of data
  - . . . so, yes, we will need this. . .

```
double *x= new double(5); // allocate space for a double, initialize it with 5
double *y=new double[5];  // allocate space of five doubles, uninitialized
x[3]=1;                   // Segmentation fault
y[3]=1;                   // Perfect...
delete x;                 // Choose the right delete!
delete[] y;               // Choose the right delete!
```

Multidimensional Arrays

- Multidimensional arrays are useful for storing matrices, tensors, arrays of coordinate vectors etc.
- It is easy to declare a multidimensional array on the stack when the size the array is known at compile time

```
double x[5][6];

for (int i=0;i<5;i++)
   for (int j=0;j<6;j++)
      x[i][j]=0;
```

- Determining array dimensions from function parameters may work with some compilers, but are not backed by the standard
- One option to have 2D arrays with arbitrary, run-time defined dimensions is to allocate a an array of pointers to double, and to use `new` to allocate each (!) row

  ... this leads to nowhere ...

Intermediate Summary

- This was mostly all (besides structs) of the C subset of C++
  - Most "old" C libraries and code written in previous versions of C++ are mostly compatible to modern C++
- Many "classical" programs use the `(int size, double * data)` style of passing data, especially in numerics
  - UMFPACK, Pardiso direct solvers
  - PetsC library for distributed linear algebra
  - triangle, tetgen mesh generators
  - ...
- On this level it is possible to call Fortran programs from C++
  - BLAS vector operations
  - LAPACK dense matrix linear algebra
  - ARPACK eigenvalue computations
  - ...
- Understanding these interfaces is the main reason to know about plain C pointers and arrays
- Modern C++ has easier to handle and safer ways to do these things, so they should be avoided as much as possible in new programs

Classes and members

- Classes are data types which collect different kinds of data, and methods to work on them.

```
class class_name
{
  private:
    private_member1;
    private_member2;
    ...
  public:
    public_member1;
    public_member2;
    ...
};
```

- If not declared otherwise, all members are private
- `struct` is the same as `class` but by default all members are public
- Accessing members of a class object:

```
class_name x;
x.public_member1=...
```

- Accessing members of a pointer to class object:

```
class_name *x;
(*x).public_member1=...
x->public_member1=...
```

Example class

- Define a class `vector` which holds data and length information and thus is more comfortable than plain arrays

```
class vector
{
  private:
      double *data;
  public:
       int size;
      double get_value( int i) {return data[i];};
      void set_value( int i, double value); {data[i]=value;};
};

...

{
  vector v;
  v.data=new double(5);  // would work if data would be public
  v.size=5;
  v.set_value(3,5);

  b=v.get_value(3); // now, b=5
  v.size=6;  // size changed, but not the length of the data array...
             // and who is responsible for delete[] at the end of scope ?
}
```

- Methods of a class know all its members
- It would be good to have a method which constructs the vector and another one which destroys it.

Constructors and Destructors

```
class vector
{ private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size(){ return size;};
      double get_value( int i ) { return data[i]; };
      void set_value( int i, double value ) { data[i]=value; };
      Vector( int new_size ) { data = new double[new_size];
                               size=new_size; };
      ~Vector() { delete [] data; };
};
...
{ vector v(5);
  for (int i=0;i<5;i++) v.set_value(i,0.0);
  v.set_value(3,5);
  b=v.get_value(3); // now, b=5
  v.size=6;  // Size is now private and can not be set;
  vector w(5);
  for (int i=0;i<5;i++) w.set_value(i,v.get_value(i));
  // Destructors automatically called at end of scope.
}
```

- Constructors are declared as `classname(...)`
- Destructors are declared as `~classname()`

Interlude: References

- C style access to objects is direct or via pointers
- C++ adds another option - references
  - References essentially are alias names for already existing variables
  - Must always be initialized
  - Can be used in function parameters and in return values
  - No pointer arithmetics with them
- Declaration of reference

```
double a=10.0;
double &b=a;

b=15; // a=15 now as well
```

- Reference as function parameter: no copying of data!

```
void do_multiplication(double x, double y, double &result)
{
    result=x*y;
}
...
double x=5,y=9;
double result=0;
do_multiplication(x,y,result) // result now contains 45
```

Vector class again

- We can define () and [] operators!

```
class vector
{
  private:
      double *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      double & operator()(int i) { return data[i]; };
      double & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new double[new_size];
                              size=new_size;}
      ~vector() { delete [] data;}
};
...
{
  vector v(5);
  for (int i=0;i<5;i++) v[i]=0.0;
  v[3]=5;
  b=v[3]; // now, b=5
  vector w(5);
  for (int i=0;i<5;i++) w(i)=v(i);
}
```

Matrix class

- We can define `(i,j)` but not `[i,j]`

```
class matrix
{ private:
      double *data=nullptr;
      int size=0;  int nrows=0;
      int ncols=0;
  public:
      int get_nrows( return nrows);
      int get_ncols( return ncols);
      double & operator()(int i,int j) { return data[i*nrow+j]);
      matrix( int new_rows,new_cols)
      { nrows=new_rows; ncols=new_cols;
        size=nrows*ncols;
        data = new double[size];
      }
      ~matrix() { delete [] data;}
};
...
{
  matrix m(3,3);
  for (int i=0;i<3;i++)
    for (int j=0;j<3;j++)
        m(i,j)=0.0;
}
```

Inheritance

- Classes in C++ can be extended, creating new classes which retain characteristics of the base class.
- The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

```
class vector2d
{  private:
    double *data;
    int nrow, ncol;
    int size;
  public:
    double & operator(int i, int j);
    vector2d(int nrow, ncol);
    ~vector2d();
}
class matrix: public vector2d
{ public:
    apply(const vector1d & u, vector1d &v);
    solve(vector1d &u, const vector1d &rhs);
}
```

- All operations which can be performed with instances of `vector2d` can be performed with instances of `matrix` as well
- In addition, `matrix` has methods for linear system solution and matrix-vector multiplication

C++: Generic programming

Generic programming: templates

- Templates allow to write code where a data type is a parameter
- We want do be able to have vectors of any basic data type.
- We do not want to write new code for each type

```
template <typename T>
class vector
{
  private:
      T *data=nullptr;
      int size=0;
  public:
      int get_size( return size);
      T & operator[](int i) { return data[i]; };
      vector( int new_size) { data = new T[new_size];
                              size = new_size;};
      ~vector() { delete [] data;};
};
...
{
  vector<double> v(5);
  vector<int> iv(3);
}
```

C++ template libray

- The standard template library (STL) became part of the C++11 standard
- Whenever you can, use the classes available from there
- For one-dimensional data, std::vector is appropriate
- For two-dimensional data, things become more complicated
  - There is no reasonable matrix class
    - std::vector< std::vector> is possible but has to allocate each matrix row and is inefficient
  - it is hard to create a std::vector from already existing data

Smart pointers

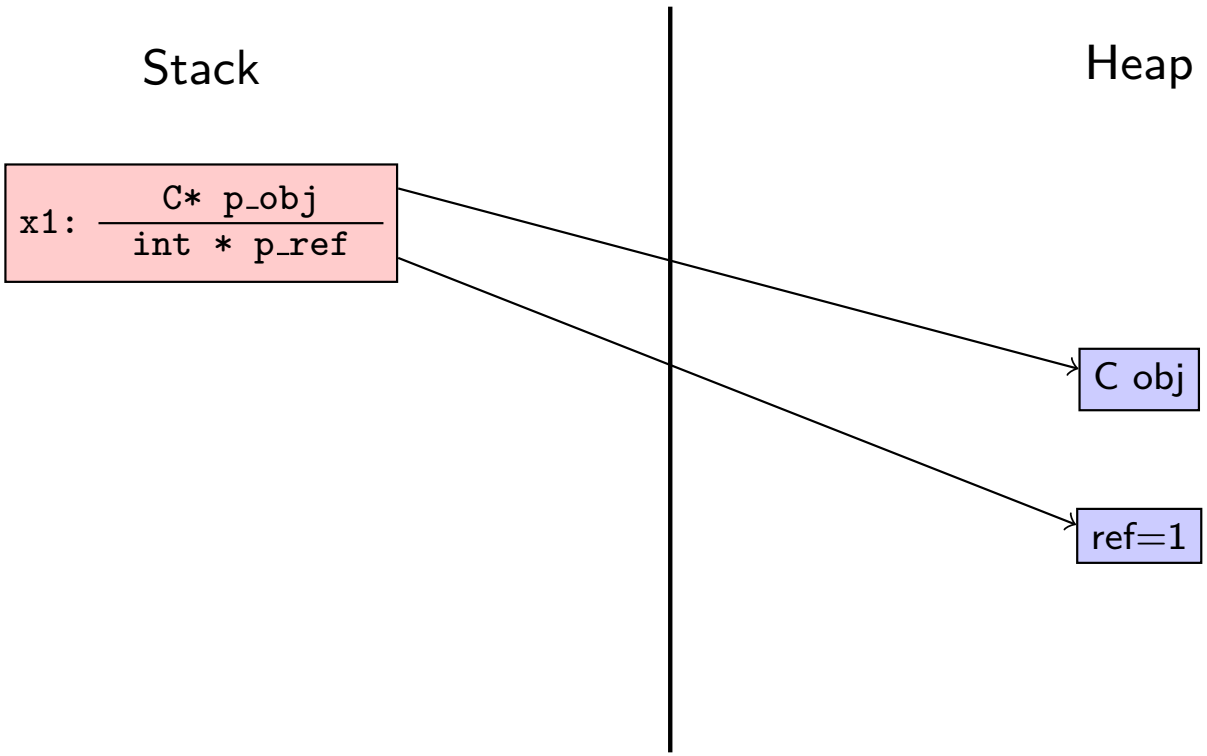
. . . with a little help from Timo Streckenbach from WIAS who introduced smart pointers into our simulation code.

- Automatic book-keeping of pointers to objects in memory.
- Instead of the meory address of an object aka. pointer, a structure is passed around *by value* which holds the memory address and a pointer to a *reference count* object.
- It delegates the member access operator `->` and the address resolution operator `*` to the pointer it contains.
- Each assignment of a smart pointer increases this reference count.
- Each destructor invocation from a copy of the smart pointer structure decreases the reference count.
- If the reference count reaches zero, the memory is freed.
- `std::shared_ptr` is part of the C++11 standard

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

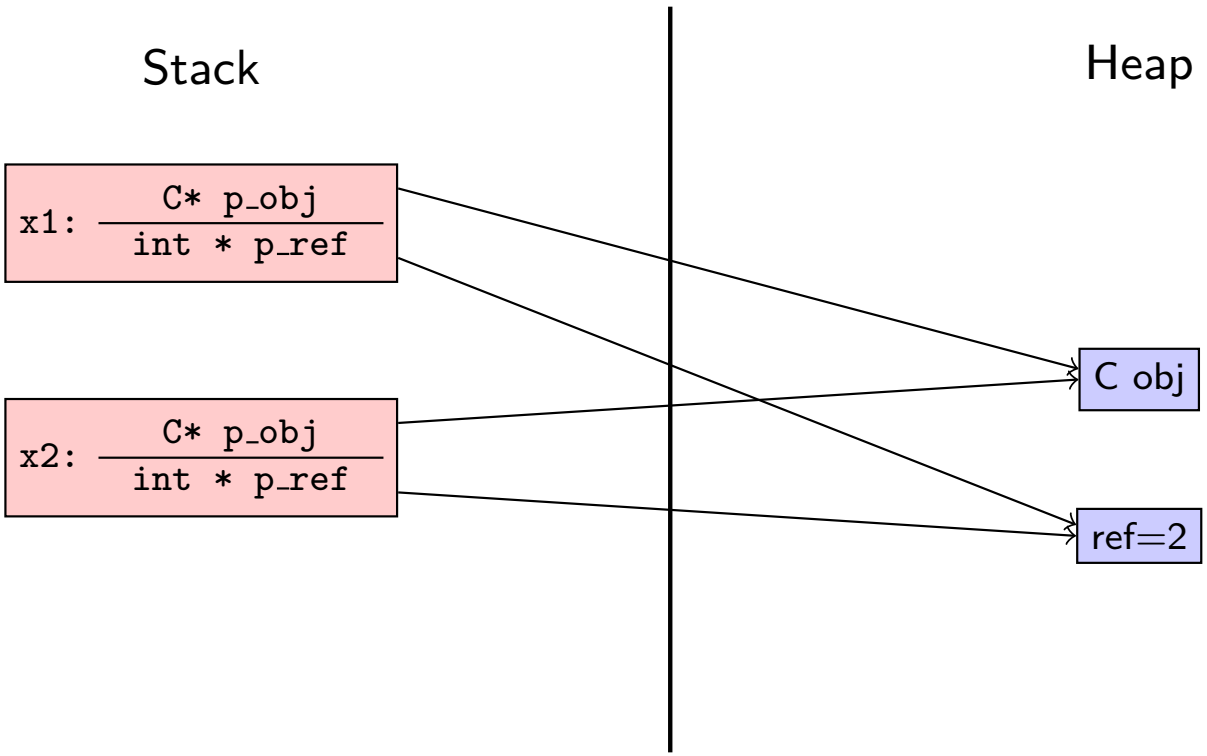



```
std::shared_ptr<C> x1= std::make_shared<C>();
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```

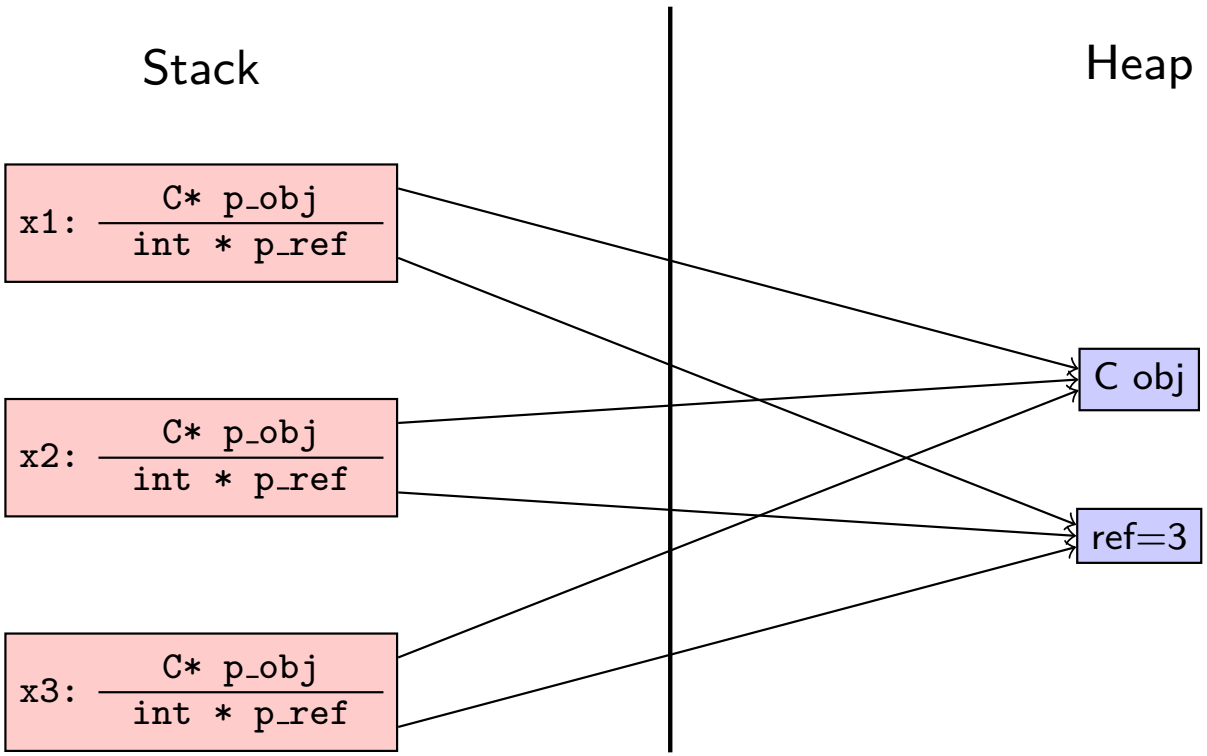



```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
```

Smart pointer schematic

(this is one possibe way to implement it)

```
class C;
```




```
std::shared_ptr<C> x1= std::make_shared<C>();
std::shared_ptr<C> x2= x1;
std::shared_ptr<C> x3= x1;
```

Smart pointers vs. *-pointers

- When writing code using smart pointers, write

```
#include <memory>
class R;
std::shared_ptr<R> ReturnObjectOfClassR(void);
void PassObjectOfClassR(std::shared_ptr<R> pR);
...
{  auto pR=std::make_shared<R>();
   PassObjectOfClassR(pR)
   // Smart pointer object is deleted at end of scope and frees memory
}
```

instead of

```
class R;
R* ReturnObjectOfClassR(void);
void PassObjectOfClassR(R* o);
...
{  R* pR=new R;
   PassObjectOfClassR(pR);
   delete pR; // never forget this here!!!
}
```

Smart pointer advantages vs. *-pointers

- "Forget" about memory deallocation
- Automatic book-keeping in situations when members of several different objects point to the same allocated memory
- Proper reference counting when working together with other libraries