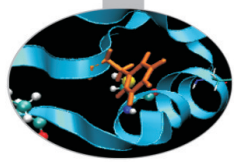
CINECA SCAI
SuperComputing Applications and Innovation

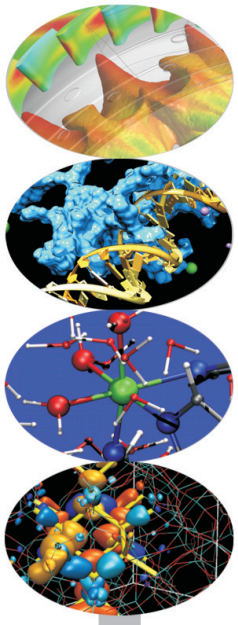


Scientific and Technical Computing in C++

Part 1 C++ - C == ?

Luca Ferraro Mario Tacconi

CINECA Roma - SCAI Department



Rome, November 2016

CINECA

Outline

C++ History

- C++ is rooted in ‘C with Classes’
  - A language developed from 1979 to 1983
  - Added Simula-like object oriented (OO) features to C
- First C++ version defined in 1984
- New versions released until an official standardization process begun in 1989
- At the same time, C++ compilers started to spread
- First Standard released in 1998 (C++ ’98)
- Revision released in 2003 (C++ ’03)
- Latest and greatest C++ Standards (C++11 & C++14)
  - approved in March 2011 and August 2014 respectively
  - they bring significant innovations
  - not covered in this course

C++ General Philosophy

- C++ is all about:
  - program structure
  - program modularization
  - program safety
  - code reuse
- An incredibly rich language, combining, at different levels, many programming paradigms:
  - procedural
  - object oriented
  - functional
  - declarative
- An incredibly rich library, providing generic data structures and algorithms commonly used in all application domains

C++: Good or Bad?

- They say C++ is bad for scientific & technical computing
  - Performance is often inferior to Fortran
  - It's a complex and difficult language
  - Prone to feature-driven abuses
  - OO programming requires very careful design
- Why C++ is good for scientific & technical computing
  - \+ It's incredibly flexible and powerful
  - \+ Allows for very high level, domain specific programming style
  - \+ GUI and DB accesses are best programmed in C++
  - \+ C++ compilers are getting better at optimizing
  - there's a steadily growing number of scientific libraries and applications written in C++
- A good language should be able to express what you need in an easy, robust, efficient way. C++ does it.

Words from the father of C++

No programming language is perfect. Fortunately, a programming language does not have to be perfect to be a good tool for building great systems. In fact, a general-purpose programming language cannot be perfect for all of the many tasks to which it is put. ... Thus, C++ was designed to be a good tool for building a wide variety of systems and to allow a wide variety of ideas to be expressed directly.
(Bjarne Stroustrup)

Our Aims

- Teach you the fundamentals of the C++ (’03) language
- For both reading and writing programs
- Showing common patterns of OO design and programming
- Illustrating best practices
- Enabling you to understand that "yet there is method in’t"
- Focusing on scientific and technical use cases

- Note: it is impossible to cover all of the language in a few days course
- Study of good books and papers, reference manuals, and personal practice are paramount in C++

Outline

Good Ol' C Style


```
/* roots of a 2nd degree equation with real coefficients */
#include <math.h>
#include <stdio.h>

int main() {
  double a, b, c, delta, x1, x2;

  printf ("Solving ax^2+bx+c=0, enter a, b, c: ");
  scanf ("%lf ,%lf ,%lf", &a, &b, &c);

  delta = b*b - 4.0*a*c;
  if (delta < 0.0) {
    fprintf (stderr, "Sorry, no real roots.\n");
    return -1;
  }
  delta = sqrt (delta);

  x1 = (-b + delta) / (2.0 * a);
  x2 = (-b - delta) / (2.0 * a);

  printf ("Real roots: %lf, %lf\n", x1, x2);
  return 0;
}
```

The C++ Style


```
/* roots of a 2nd degree equation with real coefficients */
#include <cmath>
#include <iostream>

using namespace std;

int main() {
  double a, b, c;
  cout << "Solving ax^2+bx+c=0, enter a, b, c: ";
  cin >> a >> b >> c;

  double delta = b*b - 4.0*a*c;
  if (delta < 0.0) {
    cerr << "Sorry, no real roots.";
    return -1;
  }
  delta = sqrt (delta);

  double x1, x2;
  x1 = (-b + delta) / (2.0 * a);
  x2 = (-b - delta) / (2.0 * a);

  cout << "Real roots: " << x1 << ", " << x2 << endl;
  return 0;
```

A Definite Improvement

- I/O is amazingly easier
- `cout << "Solving ax^2+bx+c=0 ...";`
  - writes the string literal to the standard output stream `std::cout`
- `cout << "Real roots:" << x1 << "," << x2 << endl;`
  - the inserter `<<` associates left to right
  - x1 and x2 are converted and concatenated
  - `endl` is much more intuitive than `'\n'`
- `cerr << "Sorry, no real roots.";`
  - writes to the standard error stream `std::cerr`
  - the extractor `>>` also associates left to right
- No format strings and conversion specifiers, at last!

More Differences

- Standard header files lost trailing **`.h`**
  - **`#include <iostream>`** provides **`cout`**, **`cin`**, **`cerr`**, as well as **`<<`**, **`>>`**, and **`endl`**
- And C standard header files got a beginning **`c`**
  - **`#include <cmath>`** gives access to functions from C **`math.h`**
  - use **`#include <cstdio>`** if you are really in love with **`printf()`** and **`scanf()`**
  - Useful to quickly port C code
- **`using namespace std;`** gives access to C++ standard library facilities (such as **`cout`**, **`cin`**, ...)
  - More on this later

Hands-on Session #1

- Write the example in a file with the `.cpp` extension and compile it using
  - `g++ find_roots.cpp`
- Play with C++ streams I/O
- Try commenting out `using namespace std;` directive
- And use the `std::` specifier where needed
- While getting acquainted with the environment, editor, compiler

Not as Complex as You Might Fear


```
#include <cmath>
#include <iostream>
#include <complex>

using namespace std;

int main () {
  double a, b, c;
  cout << "Solving ax^2+bx+c=0, enter a, b, c: ";
  cin >> a >> b >> c;

  complex<double> delta;
  delta = b * b - 4.0 * a * c;
  delta = sqrt (delta);

  complex<double> z1, z2;
  z1 = (-b + delta) / (2.0 * a);
  z2 = conj (z1);
  cout << "Complex roots: " << z1 << ", " << z2 << endl;

  return 0;
}
```

Complex Numbers

- `#include <complex>`
  - Provides complex types, operands and functions
- `complex<double> delta`
  - Quite stunning beasts at first sight!
  - Real and imaginary parts are in double precision
  - `complex<float>` and `complex<long double>` also available
- Do you want real or imaginary part?
  - Use `real(z1)` and `imag(z1)`
- No compatibility with C99 `complex.h` types
  - `double complex z1;` will be rejected
- And, again, `<<` beats `printf()` hands down

Other Things to Know About C++
`complex`

- All standard operators are defined
  - Unary operators (`+`, `-`)
  - Binary operators (`+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`, `==`, `!=`)
- And common functions too
  - `abs()` and `arg()` return modulus and argument
  - `norm()` returns square of modulus
  - Trigonometric and hyperbolic functions `sin()`, `cos()`, `tan()`, `sinh()`, `cosh()`, etc
  - And more such as `sqrt()`, `exp()`, `log()`, `pow()`
- Did you notice something is missing in functions names?
  - In C we would have to call `csqrt()`, `ctan()`...
- Generic function names, at last!

Outline

Greatest Common Divisor

- Euclid's Algorithm
  1. Take two natural numbers a and b
  2. Let $r \leftarrow a \mod b$
  3. Let $a \leftarrow b$
  4. Let $b \leftarrow r$
  5. If b is not zero, go back to step 2
  6. a is the GCD
- We want to implement it in a function
- Generalizing it to standard mathematical conventions
  - $\gcd(a, b)$ is non negative, even if a or b is less than zero
  - $\gcd(a, 0)$ is $|a|$
  - $\gcd(0, 0)$ is 0
- We want to add a least common multiple function (LCM)
- And we want it for both `int` and `long` integer types

GCD & LCM: Good Ol' C style


```
#include "numbertheory.h"

// Greatest Common Divisor
int gcd(int a, int b) {

  a = abs(a);
  b = abs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
int lcm(int a, int b) {
  if (a == 0 || b == 0)
    return 0;
  return a*(b/gcd(a,b));
}
```

GCD & LCM for `long ints`

- We could do it à la C
  - changing type to function arguments and return values
  - calling `labs()` to compute the absolute value
  - and changing the function names
- And it works...
- But C++ has a better way: function overloading!
  - Allows for function generic names to be used for different argument types
  - And is put to good use in the C++ Standard Library

GCD & LCM for `long ints`: C Style


```
// includes as before...
// gcd() and lcm() for ints as before...

// Greatest Common Divisor for long ints
long int lgcd(long int a, long int b) {

  a = labs(a);
  b = labs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    long int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple for long ints
long int llcm(long int a, long int b) {
  if (a == 0 || b == 0)
    return 0;
  return a*(b/lgcd(a,b));
}
```

GCD & LCM for `long ints`: Overloading


```
// includes as before...
// gcd() and lcm() for ints as before...

// Greatest Common Divisor
long gcd(long int a, long int b) {

  a = abs(a);
  b = abs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    long int t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
long lcm(long int a, long int b) {
  if (a == 0 || b == 0)
    return 0;
  return a*(b/gcd(a,b));
}
```

Function Overloading

- Using the same name for operations on different types is called overloading
  - function names remain the same
  - arguments differ in type and/or number
- Useful when some functions conceptually perform the same task on different types
- Standard practice for base types operators ...
  - there is only one name for addition: **+**
  - yet it can be used to add integers values, floating point values, etc
- ... and on Standard Library functions like **`sqrt`** or **`abs`**

Hands-on Session #2


```
#include "numbertheory.h"

void test_number_theory(int a, int b, long c,
                        long d, short s, unsigned u)
{
  cout << " gcd= " << gcd(a,b) << endl;
  cout << " gcd= " << gcd(c,d) << endl;
  cout << " gcd= " << gcd(a,d) << endl;
  cout << " gcd= " << gcd(c,0) << endl;
  cout << " gcd= " << gcd(c,0.0) << endl;
  cout << " gcd= " << gcd(a,s) << endl;
  cout << " gcd= " << gcd(c,u) << endl;
}
```

- Write a program:
  - including the function above:
  - and a **`main()`** calling it with suitable arguments
- You may want to add some output to each version of the function to spot which one is called

How Overloading Works

- The compiler resolves which function is to be invoked by comparing the types of the actual arguments with those of the formal parameters
- A number of criteria are tried in order:
  - Exact match, using no or only trivial conversions (`T[]` to `*T`, `T` to `const T`)
  - Match using promotions and conversions (`bool`,`short`,`char` to `int`, `float` to `double`, `double` to `long double`)
  - Match using mixed type conversions (`int` to `double`, `double` to `int`, `int` to `unsigned`)
- If two matches are found at the highest level, the call is rejected as ambiguous

Using Overloaded GCD


```
#include "numbertheory"

void test_number_theory(int a, int b, long c, long d, short s, unsigned u)
{
  cout << " gcd= " << gcd(a,b) << endl; // calls gcd(int, int)
  cout << " gcd= " << gcd(c,d) << endl; // calls gcd(long, long)
  cout << " gcd= " << gcd(a,d) << endl; // error: gcd(int,int) or gcd(long, long) ???
  cout << " gcd= " << gcd(c,0) << endl; // error: gcd(int,int) or gcd(long, long) ???
  cout << " gcd= " << gcd(c,0.0) << endl; // conversion: calls gcd(long, long)
  cout << " gcd= " << gcd(a,s) << endl; // promotion: calls gcd(int, int)
  cout << " gcd= " << gcd(c,u) << endl; // conversion: calls gcd(long, long)
}
```

- We may add an explicit type conversion to resolve a specific call, like **`gcd(long(a),d)`**
- ... or write another overloading for mixed argument types, like **`long gcd(int,long)`**
- Let's address this later on

Using Overloaded GCD

- What about overloading functions on return types?

```
int    fibonacci(int n);  // for small n values
long   fibonacci(int n);  // for larger n values
double fibonacci(int n);  // for even larger n values, with approximation
```

- No way, man ... compiler will bark!

```
fibonacci.cpp:    In function 'long fibonacci(int n)':
fibonacci.cpp:51: error: new declaration 'long fibonacci(int n)'
fibonacci.cpp:6:  error: ambiguates old declaration 'int fibonacci(int n)'
...
fibonacci.cpp:    In function 'long fibonacci(int n)':
fibonacci.cpp:95: error: new declaration 'double fibonacci(int n)'
fibonacci.cpp:6:  error: ambiguates old declaration 'int fibonacci(int n)'
```

- Return types are not taken into account to resolve overloading!

The Name Clash Problem

- Suppose we want to use our GCD implementation in an existing code and:
  - the code already makes use of an external GCD function named `gcd`
  - or has a variable named `gcd`
- One could be tempted to modify the previously used names to distinguish from ours
  - An error prone waste of time
- C++ solves the problem with namespaces

Namespaces: a Scope for Names

- Namespaces are a mechanism to express logical grouping
- Let's group functions declarations in **`numbertheory.h`** into a **`namespace`**

```
namespace numbertheory {
  long gcd(long int a, long int b);
  long lcm(long int a, long int b);
}
```

- And modify function definitions in **`numbertheory.cpp`**

```
#include "numbertheory.h"

long numbertheory::gcd(long int a, long int b) {
  //...
}

long numbertheory::lcm(long int a, long int b) {
  // ...
}
```

Accessing Names in a Namespace

- We can now access our functions like that:

```
#include <numbertheory.h>

// ...
int gcd = numbertheory::gcd(a, b);
```

- Or, when no name clash is present:

```
#include <numbertheory.h>

using namespace numbertheory;
// ...
int c = gcd(a, b);
int d = lcm(a, b);
```

- A **`using`** directive makes names from a namespace available as if they had been declared outside their namespace

The Most Important Namespace

- The **`using namespace std`** directive is a very common construct to access facilities from C++ Standard Library

```
#include <iostream>

using namespace std;

int main() {
  double a, b, c;
  cout << "Solving ax^2+bx+c=0, enter a, b, c: ";
  cin >> a >> b >> c;
  // ...
```

- Which is less tedious with respect to:

```
#include <iostream>

int main() {
  double a, b, c;
  std::cout << "Solving ax^2+bx+c=0, enter a, b, c: ";
  std::cin >> a >> b >> c;
  // ...
```

Namespaces Are Open

- We can wrap existing functions/variables inside a new namespace

- And add new members into an already existing namespace

```
#include <numbertheory.h>

namespace numbertheory {
  const long long_prime = 2147999999L;

  bool test_gcd(long a, long b) {
     return (gcd(a, b)*lcm(a, b) == a*b) &&
            (gcd(a, long_prime) == 1L) &&
            (gcd(b, long_prime) == 1L) &&
            (gcd(long_prime, a) == 1L) &&
            (gcd(long_prime, b) == 1L) ;
  }
}
```

Namespaces are Composable

- If a namespace clashes (or is too long) we can alias it:

```
namespace NT=numbertheory;
```

- Multiple default namespaces to access names from can be selected

```
using namespace numbertheory;
using namespace prime_numbers;
```

- Other namespaces can be open inside a namespace

```
namespace numbertheory {
  using namespace prime_numbers;
  //...
}
```

but then an:

```
using namespace numbertheory;
```

directive will open **`prime_numbers`** namespace too

Gaussian Distribution

- We want a function to compute the Gaussian distribution:

$$p(x;\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- And we also want the ones for the quite common special cases:

$$\sigma = 1$$

and

$$\mu = 0, \sigma = 1$$

- Easy! Overload!
- But C++ has an even easier way...

Default Arguments


```
#include <cmath>

double gaussian(double x, double mu=0.0, double sigma=1.0) {
  double pi2 = 2.0*acos(-1.0);
  double m = x - mu;
  return exp(-m*m/(2.0*sigma*sigma))/(sigma*sqrt(pi2));
}
```

- Two additional overloaded versions with only one and two arguments respectively are automatically generated
- Pay attention: it is position dependent!
  - If one argument has a default value, all following ones must have too
  - Otherwise said:

    `gaussian(double x, double mu=0.0, double sigma);`

    is forbidden

Using Templates for `gcd` & `lcm`

- Using overloading simply to change type is boring and error prone
- The algorithm is the same, only the type we work with changes

- Say something once, why say it again?

- Do it with function templates!
  - Write the function for a generic type
  - And have the compiler generate type specific versions on demand

GCD & LCM as Templates


```
// Greatest Common Divisor
template <typename Type>
Type gcd(Type a, Type b)
{

  a = abs(a);
  b = abs(b);

  if (a == 0)
    return b;
  if (b == 0)
    return a;

  do {
    Type t = a % b;
    a = b;
    b = t;
  } while (b != 0);

  return a;
}

// Least Common Multiple
template <typename Type>
Type lcm(Type a, Type b)
{
  if (a == 0 || b == 0)
    return 0;
  return a*(b/gcd<Type>(a,b));
}
```

Generic Programming

- `template <typename X>` specifies this is a template
- `typename X` declares a template parameter `X` that corresponds to any known type
  - Predefined types (`int`, `double`, etc...)
  - And user defined ones
- `X` can be used inside a template like a regular type
  - To declare variables
  - Or function arguments and return types
- To be used, template definition must be in scope!
  - That's why they are frequently put in header files
  - As we have to do with `numbertheory.h`, dispensing with numbertheory.cpp

Hands-on Session #3

- Our templates can be called as easy as **`gcd(a,b)`**
  - The template type argument will be deduced by the types of **`a`** an **`b`**
- Or the type can be explicitly specified as in **`gcd<long>(a,d)`**
  - Which could be annoying, but easier to read and write than a cast on a function argument
- When deduction is ambiguous, type specification is mandatory

- Test the new implementation on the test program you wrote before
- And play with the two forms
- Then try calling **`gcd()`** without template type specification on **`float`** or **`double`** arguments

Using templates


```
#include "numbertheory.h"

void test_number_theory(int a, int b, long c, long d, double lf)
{
  cout << " gcd= " << gcd<int>(a,b) << endl; // calls gcd(int, int)
  cout << " gcd= " << gcd(a,b) << endl; // calls gcd(int, int) by automatic argument c
  cout << " gcd= " << gcd<long>(c,d) << endl; // calls gcd(long, long)
  cout << " gcd= " << gcd(c,d) << endl; // calls gcd(long, long) by automatic argument
  cout << " gcd= " << gcd<int>(a,c) << endl; // calls gcd(int, int) convert c to int
  cout << " gcd= " << gcd<long>(b,d) << endl; // calls gcd(long, long) convert b to lo
  // misusage calls
  cout << " gcd= " << gcd(a,d) << endl; // error: gcd(int,int) or gcd(long, long) ???
  cout << " gcd= " << gcd(lf,10.0) << endl; // calls gcd(double, double) -> error
}
```

Using Templates

- Our templates can be called as `gcd<int>(a,b)` or `gcd<long>(a,d)`
  - Which could be annoying, but easier to read and write than a cast on a function argument
- Or the type can be implictly deduced as in `gcd(a,b)`
  - The template type argument will be deduced by the types of `a` an `b`
- When deduction is ambiguous, type specification is mandatory

Templates Issues

- A code calling `gcd(7000.0, 105.0)` will not compile
  - `%` operator is not defined for `doubles`
  - We are very lucky! We have been saved from `gcd()` abuse
- What if the `gcd()` implementation didn't make use of `%`?
- Beware of Frankenstein creations with templates

- Moreover, a template will be recompiled for each source file calling it
  - Complex code compilation time may grow by order of magnitudes
- Enough for now. If you're in need of preventing template instantiation for some type, or cope with mixed type return value, look for advanced techniques such as traits and concepts.

Better Than Macros

- Let's imagine we need the Heaviside function

```
double theta(double x) {
    if (x < 0.0)
      return 0.0;
    return 1.0;
}
```

- The function call costs more than its execution
- In C, we would put a macro in a header file:

```
#define theta(x) ((x) < 0.0 ? 0.0 : 1.0 )
```

- Trading readability and type checking for speed
- In C++, we can have all of them:

```
inline double theta(double x) {
    if (x < 0.0)
      return 0.0;
    return 1.0;
}
```

- Like macros, inline functions must be put in header files
- To really appreciate it, let's look at a more complicated example

Efficient C Fibonacci Implementation


```
#include<errno.h>
#include "fibonacci.h"

const unsigned int FibonacciNumbers[UINT_MAX_FIB_N+1] =
{ 0U, 1U, 1U, 2U, 3U, 5U, 8U, 13U, 21U, 34U, 55U,
  89U, 144U, 233U, 377U, 610U, 987U, 1597U, 2584U,
  4181U, 6765U, 10946U, 17711U, 28657U, 46368U,
  75025U, 121393U, 196418U, 317811U, 514229U,
  832040U, 1346269U, 2178309U, 3524578U, 5702887U,
  9227465U, 14930352U, 24157817U, 39088169U, 63245986U,
  102334155U, 165580141U, 267914296U, 433494437U, 701408733U,
  1134903170U, 1836311903U, 2971215073U
}

unsigned long fibonacci(unsigned int n) {

    if (n > UINT_MAX_FIB_N) {
      errno = ERANGE;
      return UINT_MAX;
    }
    return FibonacciNumbers[n];
}
```

Way Much Better Than Macros!

- Imagine we don't want to pay the cost of a function call
- But still, we want as much type checking as possible
- In C, we'd put in the Fibonacci header:

```
#define ONLY_POSITIVE_N_fib(n) \
                    ((n) > UINT_MAX_FIB_N ? \
                     (errno = ERANGE , UINT_MAX) \
                    : FibonacciNumbers[(n)])

#define fibonacci(n) ( (n)<0 ? (errno = EDOM, 0) :  ONLY_POSITIVE_N_fib(n) )
```

- Reader friendly, isn't it?
- In C++, instead, we can put in the Fibonacci header:

```
#include <cerrno>

inline unsigned long fibonacci(unsigned int n) {

    if (n > UINT_MAX_FIB_N) {
      errno = ERANGE;
      return UINT_MAX;
    }
    return FibonacciNumbers[n];
}
```

- Much better!

Some Remarks on Inlining

- An inline function must be:
  - in scope wherever is used
  - with identical definitions in all the program
- That's why it's usually put in a header
- Again, there is a compile time price to pay
  - Code expansion and recompilation take time
- Use inlining only where it makes sense
  - I.e. for often called, small functions

- Templates and inline functions give much more power than the preprocessor
- Preprocessor usage is explicitly discouraged in C++
  - Try to restrict its use to header file
  - And limit yourself to conditional directives

References

- Like in C, arguments to functions are passed by value
  - I.e. a copy is made
- In C, to avoid copying big structures, we'd use pointers

```
int process(const event *e) {
    // use *e and e->member like they were variables
}
```

- In C++, pass by reference is supported

```
int process(const event& e) {
    // use e and e.member as regular variables
}
```

- C++ references come in handy also as shorthands

```
double& u = grid->block[b]->fields.u[k][j][i];
double& v = grid->block[b]->fields.v[k][j][i];
double& w = grid->block[b]->fields.w[k][j][i];
double& p = grid->block[b]->fields.p[k][j][i];
double& T = grid->block[b]->fields.T[k][j][i];
// use u, v, w, p, and T as regular variables
```

- Once initialized, C++ references cannot be altered
  - All operators and functions act on the variable referred to

Enter `new` and `delete`

- Yet pointers are still of paramount importance in C++
- Particularly for dynamic memory allocation
- In C we would write:

```
signal = (data_set *)malloc(sizeof(data_set));
signal->points = n;
signal->data = (data_point *)malloc(n*sizeof(data_point));
```

and

```
free(signal->data);
free(signal);
```

- In C++ we shall write:

```
signal = new data_set;
signal->points = n;
signal->data = new data_point[n];
```

and

```
delete[] signal->data;
delete signal;
```

- Notice the specific syntax to delete an array!

The Power of `new` and `delete`

- **`new`** and **`delete`** are operators defined in **`<new>`** header
- As usual, **`malloc()`** and **`free()`** are available
- But their usage is strongly discouraged
- And forgetting them is easy because **`new`** and **`delete`** are so much handier

- We'll later realize that **`new`** and **`delete`** have more features
- Particularly in OO programming
- Addressing all their feature in an introductory course is impossible
- Their power is enough to be the single subject of an advanced course

Good Ol' Exception Handling

- Run-time exceptions may unexpectedly trash hours of already performed computations
- They must be proactively handled
- C exception handling traditionally relies on two facilities:
  - `errno`
  - special return values from Standard Library functions
- For example, in C we should write:

```
data = malloc(n*sizeof(data_point));
if (!data) {
    // possibly save already computed results
    // exit gracefully
}
```

- These facilities are still available in C++
- But C++ has a better way

The C++ Way

- In C++ we shall write:

```
try {
  data = new data_point[n];
} catch (std::bad_alloc) {
    // possibly save already computed results
    // exit gracefully
}
```

or, for I/O operations:

```
try {
  config_file >> configuration;
} catch (std::ios_base::failure) {
    // give error information to user
    // use default configuration values or exit gracefully
}
```

- Note:
  - **`std::bad_alloc`** defined in **`new`** standard header
  - **`std::ios_base::failure`** defined in **`ios`** standard header

throw

- It all works if the code in the **`try`** block *throws* exceptions on errors
- As **`new`** does
- For example, on reading in the **`key`** string variable the unknown key **`tempreature`**, the code for >> operator could:

```
throw ios_base::failure("Unknown configuration key: "+key);
```

- Exceptions are nothing more that specific C++ objects conveying informations about what happened
  - You can use the ones from the Standard Library
  - Or better define new ones according to your need, when you'll know more C++

Exception Propagation

- When a **`throw`** statement is executed:
  - the current block is exited
  - and so are the enclosing ones
  - propagating the exception object
  - until a **`try`** block is exited
- On exit from the **`try`** block:
  - the propagating exception type is matched against the **`catch`** clause
  - if a match is found, the **`catch`** block is executed
  - otherwise, the exception is re-thrown to the block enclosing the **`try`** statement and the process restart
- When the exited block is **`main()`**:
  - the exception is catched by the default catch handler
  - a message is sent to **`cerr`** and the program is terminated

Using types as error code


```
struct Zero_divide { };
struct Domain_error { };
struct Range_error { };

void compute(const Input &in, const Params &parm, Res &res) {
  if (parm.denominator == 0)
    throw Zero_divide;
  if (parm.upper>in.upper || parm.lower<in.lower)
    throw Domain_error;
  if (parm.range>in.range || parm.range<in.range)
    throw Range_error;
  // ...
}

try {
  compute(inputs, parameters, result);
} catch (Zero_divide) {
    // handle zero division
} catch (Domain_error) {
    // handle domain problems
} catch (Range_error) {
    // handle range problems
}
```

Structured Exception Handling

- With C facilities:
  - exceptions must be checked for ASAP
  - and managed right there
  - unless you want to use unstructured `longjmp()`s
- With C++ exception propagation:
  - the code is not cluttered
  - exception objects can be inspected
  - and re-thrown to an upper level if appropriate
  - until a `try` block is exited
- This is of crucial importance in OO programs, that heavily rely on composition
- Once again, covering all their features in an introductory course is impossible
- Exception handling would be a very significant part of an advanced OO design course

Outline

Recycling C code

- With some exceptions, C++ is a superset of C
- Well-written C code tends to be also C++ code
- Some incompatibilities are deemed poor style or even obsolete in modern C
- Some differences are minor
- Some differences are due to misalignment between C and C++ Standards
- Some significant differences arose as a consequence of C++ important features
- Many of them may bite you if you have to import C code in a C++ program

Identifiers and Functions

- C++ provides more reserved keywords than C does
  - Like **`public`**, **`new`**, **`class`**, **`delete`**, ...
  - These are allowed as identifiers in C, rejected by C++
- Some C Standard Library preprocessor macros are language keywords in C++
  - Like **`and`**, **`or`**, **`xor`**, **`not`**, ...
- Calling a function without a previous declaration is not allowed in C++
  - Might be encountered in very old or very bad C code
- C++ functions declared without parameters must be called without arguments
  - In C they can be called with any number and type of arguments
  - Might be encountered in very old or very bad C code

Types

- C++ **`enum`** type size is implementation defined
  - And may differ from one **`enum`** to another
  - In C **`enum`**s are **`int`**s in disguise
- Consistently, **`int`** values cannot be assigned to C++ **`enum`** variables
- C variables of any pointer type may be assigned a pointer to **`void`**
  - In C++ you'll be forced to cast correctly
  - But good programmers cast them in C too
- A C **`struct`** can have the same name of a **`typedef`** that refers to a different type
  - Not in C++
  - Might be encountered in very bad C code

Variables

- C allows for a variable to be declared without a type, and assumes it's **`int`**
  - Ugly practice disallowed in C99 and C++
  - Might be encountered in very old C code

- The following is forbidden in C++:

```
integer primes[4] = {2, 3, 5, 7, 11, 13, 17, 19};
```

  - While legal in a C program, where the last four elements will be discarded
  - But this is a suspect bug, isn't it?

Program Global Variables

- The same C file scope variable can be defined multiple times:
  - in the same source file
  - or in different source files
  - compiler and linker will sort it out
- In C++ a file scope variable can be defined only once in the program
  - I.e. you'll have to put the **`extern`** specifier to good use
- Define your variable once in a source file:

```
double tabulated_function[no_of_points];
```

- Then publish it wherever needed with a header containing the declaration:

```
extern double tabulated_function[no_of_points];
```

static

- In C, a declaration like this at file scope:
  ```
  static double table[1000];
  ```
  makes the variable invisible to other program units
- Ditto for functions, and same in C++
- However **`static`** got richer semantics in C++
- Notably, all instances of a structure declared like this:
  ```
  struct atom {
      static int count;
      int atomic_number;
      // ...
  };
  ```
  will share a single copy of **`count`** member, if defined (and possibly initialized) in a source file:
  ```
  int atom::count = 0;
  ```
- And it will be accessible to all units where the structure definition is in scope
- As a consequence, the traditional **`static`** usage is deprecated

Hiding Variables in C++

- The polite way to hide variables in C++ is an unnamed namespace:

```
namespace {
    double table[1000];
}
```

- You can put in an unnamed namespace as many variables and functions as you need
- Each unnamed namespace has an implicit **`using`** directive, so all content is in scope in the immediataly following code
- As an unnamed namespace as no name, it is impossible to access its content in other program units

The Price of Overloading

- Function overloading and templates come at a price
- To implement it, the compiler changes function names to include argument type descriptions
- So that, bottom line, each overloaded function has a unique name to the linker
- This process, termed *name mangling*, can be system or compiler dependent

- For instance, compiling with g++ version 4.7.3:
  - `int gcd(int,int)` is mangled into `_Z3gcdii`
  - `long gcd(long,long)` is mangled into `_Z3gcdll`

- This implies that functions written in different languages, like C, cannot be managed like C++ ones

The Way Around

- In order to use C external functions an **`extern "C"`** declaration must be used
  - It inhibits name mangling on affected function names
- For a single function:

```
extern "C" int init_som(som *m, vector *v);
```

- For more functions at once:

```
extern "C" {
    int init_som(som *m, vector *v);
    double train_som(som *m, vector *v, double alpha);
    // ...
}
```

- Using C linkage on a function entails obvious limitations:
  - it cannot be overloaded
  - it has to be globally visible to the linker
  - namespace control only affects C++ source compilation and has no effects to the linker

`extern "C"` Common Practices

- Need to include use a C library in a hurry?
- Use:

```
extern "C" {
    #include "multigrid.h"
}
```

- Plan to use a C library in both C and C++ programs?
- Do like in standard C header files!
- Add to header file, before all function declarations:

```
#ifdef __cplusplus
extern "C" {
#endif
```

  and after all function declarations a matching:

```
#ifdef __cplusplus
}
#endif
```

- `__cplusplus` is a preprocessor macro predefined by the C++ compiler

Table Columns Averages in C83


```
/* Use like:
   double table[N][M];
   double averages[M];
   ...
   avg(N, M, table, averages); */

void avg(int n, int m, const double *a, double *results) {
  int i,j;

  for (j=0; j<m; ++j)
    results[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<m; ++j)
      results[j] += a[i*m+j];

  for (j=0; j<m; ++j)
    results[j] /= n;
}
```

Table Columns Averages in C99


```
// Use like:
// double table[N][M];
// double averages[M];
// ...
// avg(N, M, table, averages);

void avg(int n, int m, const double a[n][m], double b[m]) {
  int i,j;

  for (j=0; j<m; ++j)
    b[j] = 0;

  for (i=0; i<n; ++i)
    for (j=0; j<m; ++j)
      b[j] += a[i][j];

  for (j=0; j<m; ++j)
    b[j] /= n;
}
```

No VLAs in C++

- C99 introduced Variable Length Arrays (VLAs)
- A feature Fortran had from ~30 years
- That makes function operating on arrays of arrays (of arrays...) much more natural to write and read

- C++ has no VLA (and neither will in the future)
  - One has to step back to the C83 version: it works
  - Or use more powerful C++ facilities

Enter C++ valarrays


```
#include<valarray>

using namespace std;

void avg(int n, int m, const valarray< valarray<double> >& a,
                                        valarray<double>& results) {

    results = 0.0;              // all elements are zeroed

    for (int i=0; i<n; i++)
        results += a[i];        // memberwise sum

    results /= n;               // memberwise division by scalar
}
```

valarrays

- STL **valarrays** are template array types
  - Designed for numeric computations
  - Supporting arithmetic on arrays as a whole
  - Much like with Fortran array syntax
- Assignment and other operators can be applied to the whole array at once
- In C tradition, one dimensional, but easily composable
- Did you get it?
  - They are templates
  - And **complex<float>** is too
  - C++ types can be templates as well

More `valarray` Features

- Sizing and initialization are easy

```
valarray<float> v0(1000);       // 1000 floats == 0.0F
valarray<int>   v1(-1,1000);    // 1000 ints == -1
valarray<int>   v2 = -2*v1;     // 1000 ints == 2
const double vd[] = {0.0, 1.0, 2.0, 3.0, 4.0};
valarray<double> v3(vd,4);      // 4 elements: 0,1,2,3
int integers[]={1,2,3,4,5,6,7,8,9,10};
valarray<int> v4(integers,10);
```

- Can be sliced, pretty much like Fortran array sections

```
valarray<double> even_numbers = v4[slice(1,v4.size()/2,2)];
```

- Functions of the base type can be mapped elementwise

```
valarray<double> v5 = v3.apply(cos); // applies cos()
                                     // to each element
```

- `v1.size()`? `v3.apply(cos)`? That's class!

Dispensing With `n` and `m`


```
#include<valarray>

using namespace std;

void avg(const valarray< valarray<double> >& a,
                         valarray<double>& results) {

    results = 0.0;              // all elements are zeroed

    int size = a.size();        // return number of elements

    for (int i=0; i<size; i++)
        results += a[i];        // memberwise sum

    results /= size;            // memberwise division by scalar
}
```

Making `avg()` as Generic as Possible


```
#include<valarray>

using namespace std;

template <typename T>
void avg(const valarray< valarray<T> >& a,
                         valarray<T>& results) {

    results = 0.0;              // all elements are zeroed

    int size = a.size();        // return number of elements

    for (int i=0; i<size; i++)
        results += a[i];        // memberwise sum

    results /= size;            // memberwise division by scalar
}
```

Enough for now

- Our C++ implementation of **`avg()`** is:
  - more readable than the C83 version
  - more compact than the C99 version
  - more generic than both of them
  - as fast as one should expect
- Let's move on to more C++ features ...

Outline

Null Terminated Strings

- C uses null terminated strings
  - Represented as array of **`chars`**
  - End marked by a **`'\0'`** terminating characters
- They have both pros and cons
  - \+ Very efficient for some operations
  - \+ Map very well to hardware architectures
  - – Very slow for others (notably **`strlen()`**)
  - – Support of non-US character sets is cumbersome
  - – Arrays fixed size is a common source of buffer overflows
- C++ still supports them
  - All related functions accessible through **`cstring`** header
- But offers a much more usable, if slightly less efficient, alternative

Easy to Use

- Declaration and initialization

```
string s = "C++ strings"; // initialized string of chars
string star(20, '*');     // 20 copies of '*' character
string r;   // an empty string
```

- Assigment and concatenation

```
r = "resize";
s += " automatically";
star = s + ' ' + r; // mixing strings with chars
```

- I/O

```
cout << star << ", no more hassles!" << endl
cin >> r;
```

- Did you get it? They automatically resize!
  - Thus slightly slower than array if **`chars`**
  - But way much easier and safer!
- Want to know the length?
  - **`r.length()`** or **`r.size()`** will do
  - In constant time!

Porting C Style Code

- Single characters in a string can be indexed by an integer of **`size_type`** type, ranging from **`0`** to **`s.length()-1`**
- Have to port a C code in a hurry? Easily done:

```
if (isalpha(c = s[i])) // build an histogram of chars
    ++histo[s[i]];     // in a string
```

  but should you go out of bounds, you'll be on your own
- The following is safer

```
if (isalpha(c = s.at(i)))  // throws std::out_of_range
    ++histo[s.at(i)];      // on out-of-bounds access
```

- Need to convert to a C string? Easily done:

```
test = strcmp(s.c_str(), star.c_str());
```

- But it's easy to switch to C++ style:

```
test = s.compare(star);
```

- And you might really like **`==`**, **`!=`**, **`>`**, **`<`**, **`>=`**, and **`<=`**
  - Which also compare **`string`**s against arrays of **`char`**s

More `string` Methods

Method	Does
`string substr(size_type b, size_type e) const;`	returns substring going from position `b` to position `e`
`string insert(size_type pos, const string& s);`	inserts `s` before `pos`, probably changing length
`size_type find(const string& s, size_type pos=0) const;`	finds first occurrence of `s` starting at `pos`
`size_type rfind(const string& s, size_type pos=0) const;`	finds first occurrence of `s` going backward from `pos`
`size_type find_first_of(const string& s, size_type pos=0) const;`	finds first occurrence of any character in `s`
`size_type find_last_of(const string& s, size_type pos) const;`	finds last occurrence of any character in `s`
`size_type find_first_not_of(const string& s, size_type pos=0) const;`	finds first occurrence of any character not in `s`
`size_type find_last_not_of(const string& s, size_type pos) const;`	finds last occurrence of any character not in `s`
`string replace(size_type pos, size_type n, const string& s);`	replaces `n` characters starting from `pos` with `s`, possibly changing length

- Call them on an object like `star.find(r)`
- `const` after function definition means the object is not modified
- For exact behavior and default arguments, browse a C++ reference
- For even more variations, browse a C++ reference

More Strings

- **`wstring`** is a string of wide characters
  - Good to cope with most languages on Earth
  - Supporting the same functionalities of **`string`**
- As usual, these are **`typedef`**ed template types
- The underlying template is **`std::basic_string`**
  - Somewhat complex
  - But may be used to create even more powerful strings
- If you process textual data in local languages, dates, ...
  - You'll better learn about the helpful **`std::locale`** class
  - But we don't cover in this course

iostream

- Streams are meant to convert internal binary data formats into/from human readable sequences of characters
- Accessible through **`<iostream>`** header
- A **`using namespace std;`** directive will make their usage more natural
- **`ostream`** is the type used for output streams
  - Like predefined **`cout`** and **`cerr`**
- **`istream`** is the type used for input streams
  - Like predefined **`cin`**
- Most I/O is performed using << and >>
  - Formally termed *inserter* and *extractor*
  - *Put to* and *get from* among friends

fstream

- **fstream** header is used to access files
- **ifstream** and **ofstream** are used for input and output respectively
- **fstream** type can be both read from and written to

- Files are opened as easily as:

```
string Tprobes_out = "temperature_probes.dat";
ifstream input("../config.dat");
ofstream output("results.dat");
ofstream *Tprobes = new ofstream(Tprobes_out.c_str());
```

- Files are closed when the object goes out of scope or, if dynamically allocated, deleted
- But you can close them earlier, if you prefer, like:

```
input.close();
```

- Everithing else is the same

Binary I/O I

- To open files in binary mode I/O manipulator must be used

```
fstream binary_file
bin_file.open("data.bin",ios::in|ios::out|ios::binary);
```

- and moving around

```
istream& seekg(streampos pos);
ostream& seekp(streampos pos);
streampos ostream::tellp();
streampos istream::tellg();
// moving at the beginning of file
bin_file.seekg (0, ios::beg);
```

- reading and writing interfaces are pretty simple
  - data handle must be of type **`char *`**, i.e. one byte pointer arithmetic

```
bin_file.write(reinterpret_cast<char *>(&my_struct), sizeof(my_struct));
bin_file.read(reinterpret_cast<char *>(&my_struct), sizeof(my_struct));
```

Binary I/O II

- use **`reinterpret_cast`** sparingly, it could be dangerous!
  - it forces the exchange between completely unrelated types
  - their use is necessary in very few cases, e.g. very low level data handling such as binary I/O
  - C cast can act as **`reinterpret_cast`** in some cases, so don't use it
- Use good ol' C binary file I/O library functions
  - Accessible through **`cstdio`** header
- Or more sophisticated alternatives
  - Like HDF5, with its nice, object oriented C++ API
  - Or more general purpose databases
- To dump and retirieve objects and complex data structures use the boost serilization library

Stream State

- Every stream has logical states
- Which can be queried using some **`bool`** methods
- If the last I/O operation on stream **`data`** succeeded:
  - **`data.good()`** returns **`true`**
  - and next operation might succeed too
- If the last I/O operation failed:
  - **`data.fail()`** returns **`true`**
  - and **`(data)`** is zero, i.e. false
  - next operation will fail too
  - characters may have lost but the stream is otherwise uncorrupted
- When the stream got corrupted:
  - **`data.bad()`** returns **`true`**
- When end of file was reached:
  - **`data.eof()`** returns **`true`**

- Remember: each `<<` or `>>` in a chain is a separate I/O operation!

I/O Manipulators

- The most frequently used:

```
cout << endl;
```

- Another one good to know:

```
cout << flush;
```

  - Forces emission of buffered text
  - Not needed on **`cerr`**

- Most manipulators are accesible through the **`<ios>`** and **`<iomanip>`** headers
- WARNING: formatting state of a stream is stateful!
  - which means we are free from abstruse format strings
  - but also implies you must not forget you set it elsewhere

Output Formatting Manipulators

- All types:

```
cout << setw(12); // NEXT output emits exactly 12 characters
cout << left;     // left justified output
cout << right;    // right justified output, default
```

- All numeric types:

```
cout << showpos;   // do emit + sign for positive numbers
cout << noshowpos; // do not, default
```

- Integer types:

```
cout << dec; // base 10 output, default
cout << hex; // base 16 output
cout << bin; // base 2 output
```

- Floating point types:

```
cout << setprecision(9); // 9 significant digits
cout << setprecision(6); // 6 digits, bad default
cout << fixed;           // dd.dddd format
cout << scientific;      // d.dddddEdd format
cout << uppercase;       // uppercase E in scientific fmt
cout << nouppercase;     // lowercase e, default
cout << showpoint;       // do print trailing 0s
cout << noshowpoint;     // do not, default
```

Input

- Like in C **`scanf()`**, input is very simple:
  1. whitespace is skipped, by default
  2. then characters are swallowed until an incompatible one or whitespace is encountered
- Manipulators:

```
cin >> setw(12); // NEXT input no more than 12 characters
                 // or 11 for array of chars
cin >> noskipws;     // don't skip whitespace
cin >> skipws;       // skip it, default
```

- If input fails, the variable stays unchanged!
  - Check stream state, if you care
- Like with C **`scanf()`**, keyboard input can be troublesome
  - Read in a whole line with **`getline(cin,`*`string_var`*`)`**
  - Then parse it

I/O on `strings`

- It is sometimes very useful to convert binary data into a string
- Or convert to binary data from a string, as we said
- the **<sstream>** header file provides:
  - **istringstream** to convert from a string
  - **ostringstream** to convert to a string
  - **stringstream** for both conversions
- Declaration:

```
string x_axis_caption;
ostringstream x_caption(x_axis_caption);
```

- All stream functionalities will work as usual

I/O on strings

- The purposes of **stringstream**s family is to provide functionality of good 'ld C **sprintf** and **sscanf** functions

```
string filename = "restart";
int index;

// ..

// composing filename as name.NNN.txt
ostringstream osfilename;
osfilename << filename << "." << setfill('0')
  << index << ".txt";

write_restart_file(osfilename.str(), program_status);
```

- which is equivalent to the following C code

```
// composing filename as name-NNN.txt in C
char filename[FILENAME_MAX_LEN];
sprintf(filename, "%s-%03d.txt", name, index);

write_restart_file(filename, program_status);
```

Want to Know More?

- You can write your own I/O operators:
  - We'll address this later
- Streams are also available for 'wide characters'
- Stream state can be set and cleared
- More manipulators are available
- And you may want to write your own
- Or exploit I/O exceptions
- Or define a new flavor of stream
- To seamlessly manage data acquisition from HEP detectors in your lab
- We cannot go to this level of detail
  - But it's perfectly feasible
  - If you learn more about streams

xyzinput.dat


```
100

B    -9.71868624e-04   -3.06242577e-03    7.12290507e-03
Si    2.70788718e+00    2.71177311e+00    1.97612206e-03
Si    2.71216726e+00    4.25709686e-03    2.70758408e+00
Si    4.78106353e-03    2.70830095e+00    2.71046762e+00
Si    1.32708325e+00    1.32608649e+00    1.32612953e+00
Si    4.07630077e+00    4.07937686e+00    1.36488865e+00


...

Si    4.10471301e+00    1.22806988e+01    9.53474911e+00
B     6.81357402e+00    1.49903747e+01    9.53953861e+00
Si    5.45459589e+00    1.09054856e+01    1.08985422e+01
Si    8.14016849e+00    1.36656210e+01    1.08642081e+01
Si    8.17440081e+00    1.09127968e+01    1.36189883e+01
Si    5.45231871e+00    1.36305778e+01    1.36234531e+01

      2.72549970e+01    0.00000000e+00    0.00000000e+00
      0.00000000e+00    2.72549970e+01    0.00000000e+00
      0.00000000e+00    0.00000000e+00    2.72549970e+01
```

Hands-on Session #5

- Write a program that:
  - asks for an xyz-format file name
  - reads it with standard input operator
  - than writes it to **`cout`**
- You may use scalar variables
- Or arrays
- Or define a **`struct`** to represent each atom and make an array thereof
- The important things to try are:
  - manipulators to control output format
  - file names that do not exist
  - files in the wrong format
  - files with missing data

Rights & Credits

Slides and examples were authored by:

- Michela Botti
- Federico Massaioli
- Luca Ferraro
- Stefano Tagliaventi