Scientific Computing (Phys 2109/Ast 3100H)
I. Scientfic Software Development

SciNet HPC Consortium

University of Toronto

Winter 2014

SciNet compute • calcul CANADA

Lecture 2

SciNet
compute • calcul
CANADA

C++ Intro: Basic syntax aspects

- Other C++ files can be included with the `#include` directive.
- Each executable statement or declaration ends with a semicolon.
- Curly braces delimit a code block.
- When declaring a variable or function to be of a certain type, the type is specified before the variable or function name.
- The value to be given back by a function is specified by the `return` statement, which exits the function.
- Comments can be added using the double slashes `//`.

C++ Intro: Variable definition

```
type name [= value];
```

Here, *type* may be a

C++ Intro: Variable definition

type *name* [= *value*];

Here, *type* may be a

* floating point type:
  `float`, `double`, `long double`, `std::complex<float>`, ...
* integer type:
  [`unsigned`] `short`, `int`, `long`, `long long`
* character or string of characters:
  `char`, `char*`, `std::string`
* boolean:
  `bool`
* array, pointer
* class, structure, enumerated type, union

C++ Intro: Variable definition

type name [= *value*];

Here, *type* may be a

* floating point type:
  `float`, `double`, `long double`, `std::complex<float>`, ...
* integer type:
  [`unsigned`] `short`, `int`, `long`, `long long`
* character or string of characters:
  `char`, `char*`, `std::string`
* boolean:
  `bool`
* array, pointer
* class, structure, enumerated type, union

Non-initialized variables are not 0, but have random values!

C++ Intro: Functions

Function declaration (prototype)

returntype *name*(*argument-spec*);

argument-spec = comma separated list of variable definitions (may be empty)

e.g.: `int main(int argc, char ** argv);`

C++ Intro: Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

argument-spec = comma separated list of variable definitions (may be empty)

e.g.: `int main(int argc, char ** argv);`

Function definition

```
returntype name(argument-spec) {
    statements
    return expression-of-type-returntype;
}
```

C++ Intro: Functions

Function declaration (prototype)

```
returntype name(argument-spec);
```

argument-spec = comma separated list of variable definitions (may be empty)

e.g.: `int main(int argc, char ** argv);`

Function definition

```
returntype name(argument-spec) {
    statements
    return expression-of-type-returntype;
}
```

Function call

```
var=name(argument-list);
f(name(argument-list));
name(argument-list);
```

argument-list = comma separated list of values

C++ Intro: Namespaces

- Variables and function, as well as variable types, have names.
- In larger projects, name clashes can occur.
- Solution: put all functions, types, ... in a namespace:

```
namespace nsname {
...
}
```

- Effectively prefixes all of `...` with *nsname::*
  Example:
  `std::cout << "Hello, world" << std::endl;`
- Many standard functions/types are in namespace std.
- To omit the prefix, do using namespace *nsname*;
- Can selectively omit prefix, e.g., `using std::vector;`

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passval passval.cc
$
```

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passval passval.cc
$./passval
```

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passval passval.cc
$./passval
$
```

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passval passval.cc
$./passval
$ echo $?
```

C++ Intro: Pass by value or by reference

Passing function arguments (default)

```
// passval.cc
void inc(int i) {
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passval passval.cc
$./passval
$ echo $?
10
$
```

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passref.cc
void inc(int &i){
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passref.cc
void inc(int &i){
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passref passref.cc
$
```

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passref.cc
void inc(int &i){
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passref passref.cc
$ ./passref
$
```

C++ Intro: Pass by value or by reference

Passing function arguments by reference

```
// passref.cc
void inc(int &i){
   i = i+1;
}
int main() {
   int j = 10;
   inc(j);
   return j;
}
```

```
$ g++ -o passref passref.cc
$ ./passref
$ echo $?
11
$
```

C++ operators

Arithmetic

`a+b`	Add a and b
`a-b`	Subtract a and b
`a*b`	Multiply a and b
`a/b`	Divide a and b
`a%b`	Remainder of a over b

Logic

`a==b`	a equals b
`a!=b`	a does not equal b
`!a`	a is not true (also: `not a`)
`a&&b`	both a and b true (also: `a and b`)
`a||b`	either a or b is true (also: `a or b`)

Assignment

`a=b`	Assign an expression b to the variable b
`a+=b`	Add b to a (result stored in a)
`a-=b`	Subtract b from a (result stored in a)
`a*=b`	Multiply a with b (result stored in a)
`a/=b`	Divide a by b (result stored in a)
`a++`	Increase value of a by one
`a--`	Decrease value of a by one

What is 1/4?

What is 1/4?

```
1/4 = 0
```

What is 1/4?

```
1/4 = 0
```

Why?

- Literal expressions, such as
  `"Hi", 0, 1.2e-4, 2.4f, 0xff, true`
  have types, just as variables do.
- The result-type of an operator depends on that of the operands.
- In "1/4" both operands are integers
- Hence, the result of 1/4 is the integer part of the division, which is 0.

To fix it, we need to be able to convert between types. In C/C++ this is called casting.

Casting one numeric type into another

Treat the type as a function.

E.g.

```
// 1over4.cc
#include <iostream>
int main() {
   int a = 1;
   int b = 4;
   int c = a/b;
   float d = float (a)/float (b);
   std::cout
      << c << " "
      << d << " "
      << int (d) << std::endl;
}
```

```
$ g++ 1over4.cc -o 1over4
$ ./1over4
0 0.25 0
```

Automatic Casting

If an expression expects a variable or literal of a certain type, but it receives another, C++ may be able to convert it automatically.
E.g.

```
1.0/4
```

is equal to

```
1.0/4.0
```

The expression mayu be a function call too, so that in

```
int unchanged(int i) {
   return i;
}
int main() {
   return unchanged(2.3);
}
```

the argument 2.3 gets converted to an int first, and then passed to the function unchanged, so the returned value is 2.

C++ Intro: Loops

```
for (initialization; condition; increment) {
    statements
}
```

```
while (condition) {
    statements
}
```

You can use `break` to exit the loop.

C++ Intro: Loops

Example

```
// count.cc
#include <iostream>
int main() {
   for (int i=1; i<=10; i++)
      std::cout << i << " ";
      // look, no braces!
   std::cout << std::endl;
}
```

C++ Intro: Loops

Example

```
// count.cc
#include <iostream>
int main() {
   for (int i=1; i<=10; i++)
      std::cout << i << " ";
      // look, no braces!
   std::cout << std::endl;
}
```

```
$ g++ -o count count.cc -O2
$ ./count
1 2 3 4 5 5 6 7 8 9 10
$
```

C++ Intro: Pointers

- Pointers are essentially memory addresses of variables.
- For each type of variable *type*, there is a pointer type *type** that can hold an address of such a variable.
- Useful in arrays, linked lists, binary trees, ...
- Null pointer, denoted by 0, points to nowhere.

C++ Intro: Pointers

- Pointers are essentially memory addresses of variables.
- For each type of variable *type*, there is a pointer type *type** that can hold an address of such a variable.
- Useful in arrays, linked lists, binary trees, ...
- Null pointer, denoted by 0, points to nowhere.

Definition:

```
type *name;
```

Assignment ("address-of"):

```
name = &variable-of-type;
```

Deferencing ("content-at"):

```
variable-of-type = *name;
```

C++ Intro: Pointers

Example

```
// ptrex.cc
#include <iostream>
int main() {
   int a = 7, b = 5;
   int *ptr = &a;
   a = 13;
   b = *ptr;
   std::cout<< "b=" << b << std::endl;
}
```

C++ Intro: Pointers

Example

```
// ptrex.cc
#include <iostream>
int main() {
   int a = 7, b = 5;
   int *ptr = &a;
   a = 13;
   b = *ptr;
   std::cout<< "b=" << b << std::endl;
}
```

```
$ g++ -o ptrex ptrex.cc -O2
$ ./ptrex
```

C++ Intro: Pointers

Example

```
// ptrex.cc
#include <iostream>
int main() {
   int a = 7, b = 5;
   int *ptr = &a;
   a = 13;
   b = *ptr;
   std::cout<< "b=" << b << std::endl;
}
```

```
$ g++ -o ptrex ptrex.cc -O2
$ ./ptrex
b=13
$
```

C++ Intro: Automatic arrays

type *name*`[`*number*`];`

- *name* is equivalent to a pointer to the first element.
- Usage: *name*`[i]`. Equivalent to `*(`*name*`+i)`.
  This is really a just a different way to dereference a pointer.
- C/C++ arrays are zero-based.

C++ Intro: Automatic arrays

Example

```
// autoarr.cc
#include <iostream>
int main() {
   int a[6]={2,3,4,6,8,2};
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
}
```

C++ Intro: Automatic arrays

Example

```
// autoarr.cc
#include <iostream>
int main() {
   int a[6]={2,3,4,6,8,2};
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
}
```

```
$ gcc -o autoarr autoarr.cc -O2
$ ./autoarr
25
$
```

SciNet compute • calcul CANADA

C++ Intro: Automatic arrays

Example

```
// autoarr.cc
#include <iostream>
int main() {
   int a[6]={2,3,4,6,8,2};
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
}
```

B A D !!

(in general)

```
$ gcc -o autoarr autoarr.cc -O2
$ ./autoarr
25
$
```

SciNet compute • calcul CANADA

C++ Intro: Automatic arrays

Example

```
// autoarr.cc
#include <iostream>
int main() {
   int a[6]={2,3,4,6,8,2};
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
}
```

B A D !!

(in general)

```
$ gcc -o autoarr autoarr.cc -O2
$ ./autoarr
25
$
```

Gotcha:

- C standard only says at least one array of at least 65535 bytes.
- In practice, limit is set by compiler and stack size.

SciNet compute • calcul CANADA

C++ Intro: Dynamically allocated array

A dynamically allocated arrays is defined as a pointer to memory:

```
type *name;
```

Allocated using the keyword `new` :

```
name = new type [number];
```

Deallocated by a function call:

```
delete [] name;
```

- Usage of these arrays is the same as for automatic arrays.
- Can access all available memory.
- Can control when memory is gived back.
- Unfortunately, no multi-dimensional version in the standard.

Improved version

Example

```
// dynarr.cc
#include <iostream>
int main() {
   int *a = new int [6];
   for (int i=0;i<6;i++){
      a[i]=i+2;
      if (i>=3)
         a[i] = 2*i*(15-2*i)-48;
   }
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
   delete [] a;
}
```

Improved version

Example

```
// dynarr.cc
#include <iostream>
int main() {
   int *a = new int [6];
   for (int i=0;i<6;i++){
      a[i]=i+2;
      if (i>=3)
         a[i] = 2*i*(15-2*i)-48;
   }
   int sum=0;
   for (int i=0;i<6;i++)
      sum += a[i];
   std::cout << sum << std::endl;
   delete [] a;
}
```

```
$ gcc -o dynarr dynarr.cc -O2
$ ./dynarr
25
$
```

C++ Intro: Dynamically allocated arrays

Example

C++ Intro: Dynamically allocated arrays

Example

```
// dyna.cc
#include <iostream>
void printarr(int n, int *a)
{
   for (int i=0;i<n;i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}
int main()
{
   int n = 100;
   int *b = new int [n];
   for (int i=0;i<n;i++)
      b[i]=i*i;
   printarr(n,b);
   delete [] b;
   return 0;
}
```

C++ Intro: Dynamically allocated arrays

Example

```
// dyna.cc
#include <iostream>
void printarr(int n, int *a)
{
   for (int i=0;i<n;i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}
int main()
{
   int n = 100;
   int *b = new int [n];
   for (int i=0;i<n;i++)
      b[i]=i*i;
   printarr(n,b);
   delete [] b;
   return 0;
}
```

```
$ g++ -o dyna dyna.cc -O2
$ ./dyna
```

C++ Intro: Dynamically allocated arrays

Example

```
// dyna.cc
#include <iostream>
void printarr(int n, int *a)
{
   for (int i=0;i<n;i++)
      std::cout << a[i] << " ";
   std::cout << std::endl;
}
int main()
{
   int n = 100;
   int *b = new int [n];
   for (int i=0;i<n;i++)
      b[i]=i*i;
   printarr(n,b);
   delete [] b;
   return 0;
}
```

```
$ g++ -o dyna dyna.cc -O2
$ ./dyna
0 1 4 9 16 25 36 49 64 81
100 121 144 169 196 225
256 289 324 361 400 441 484
529 576 625 676 729 784 841
900 961 1024 1089 1156 1225
1296 1369 1444 1521 1600
1681 1764 1849 1936 2025
2116 2209 2304 2401 2500
2601 2704 2809 2916 3025
3136 3249 3364 3481 3600
3721 3844 3969 4096 4225
4356 4489 4624 4761 4900
5041 5184 5329 5476 5625
5776 5929 6084 6241 6400
6561 6724 6889 7056 7225
7396 7569 7744 7921 8100
8281 8464 8649 8836 9025
9216 9409 9604 9801
```

Dynamic allocation of single variables

One can also dynamically allocate a single variable:

```
double * a = new double;
*a = 4;
std::cout << a << std::endl;
delete a;
```

Note the absence of `[]`.

You may use this in more dynamic data structures.

C++ Intro: Conditionals

```
if (condition) {
    statements
} else if (other condition) {
    statements
} else {
    statements
}
```

Example

C++ Intro: Conditionals

```
if (condition) {
   statements
} else if (other condition) {
   statements
} else {
   statements
}
```

Example

```
int main(){
   int n = 20;
   int *b = new int [n];
   if (b == 0)
      return 1; //error
   else {
      for (int i=0;i<n;i++)
         b[i] = i*i;
      printarr(n,b);
      delete [] b;
   }
}
```

SciNet compute • calcul CANADA

C++ Intro: Conditionals

```
if (condition) {
   statements
} else if (other condition) {
   statements
} else {
   statements
}
```

Example

```
int main(){
   int n = 20;
   int *b = new int [n];
   if (b == 0)
      return 1; //error
   else {
      for (int i=0;i<n;i++)
         b[i] = i*i;
      printarr(n,b);
      delete [] b;
   }
}
```

```
$ g++ -o ifm ifm.cc -O2
$ ./ifm
0 1 4 9 16 25 36 49 64 81 100
121 144 169 196 225 256 289
324 361
$
```

C++ Intro: Const

A type modifier

- `const` is a type modifier.
- It means the value of that type is fixed.
- Useful for constants, e.g.

```
const int arraySize = 1024;
```

- Useful to show read-only arguments to functions:

```
int f( const Type &in, Type &out );
```

- `const` is contageous!
- Now everything has to be "const correct".

C++ Intro: Objects

Object oriented programming (OOP)

- **Non-OOP:** functions and data accessible from everywhere.
- **OOP:** Data and functions (methods) together in an object. Implementation details hidden.

What are classes?

- Classes are to objects what types are to variables.
- Using a class, one can create one or more instances of it, called objects:

```
classname objectname(arguments);
```

C++ Intro: Classes and objects

Syntax:

```
classname objectname(arguments);
```

Usage

- Different from regular variables are the possibility of argument, supplied to construct the object.
- An object has members (fields) and member functions (methods), which are accessed using the "." notation.

```
object.field;
object.method(arguments);
```

C++ Intro: Classes and objects

Example (member function/method)

```
#include <string>
std::string s("Hello");
int stringlen=s.size();
```

Example (member/field)

```
#include <utility>
std::pair<int,float> p(1, 0.314e01);
int   int_of_pair   = p.first;
float float_of_pair = p.second;
```

C++ Intro: Templates

Templates

- Some algorithms are type-independent, and can be expressed with the same code, except for a change in type.
- Using *generic programming*, you write this code once, with a generic type placeholder. Versions of this code for specific types are instantiated by the compiler when needed.
- In C++, generic programming uses templates.
- Many templated functions and classes in the standard library.

C++ Intro: Template functions

- Imagine you had two functions to compute the square of a number, one for `int` s and one for `double` s:

```
int sqrit(int x) { return x*x; }
double sqrit(double x) { return x*x; }
```

  Note that they have the same name: this is fine in C++, the type of arguments in a call will determine which is used.

- We would have to write one of these functions for each type.
- Templates allow us to write the generic function just once:

```
template<typename TYPE>
TYPE sqrit(TYPE x) { return x*x; }
```

- Wherever the compiler sees a function call `sqrit(...)` it generates the right function.
- Where there's ambiguity, you can be more explicit, e.g. `sqrit<double>(3);`

C++ Intro: Class Templates

Usage

- To create an object from a template class *templateclass*:

```
templateclass<type> object(arguments);
```

Examples:

```
std::complex<float> z; //single precision complex number
std::vector<int> i(20);//array of 20 integers
```

C++ Intro: Libraries

Usage

- Put an include line in the source code, e.g.

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

- Include the libraries at link time using `-l[libname]`. Implicit for the standard libraries.

C++ Intro: Libraries

Usage

- Put an include line in the source code, e.g.

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

- Include the libraries at link time using `-l[libname]`. Implicit for the standard libraries.

Common standard libraries (Standard Template Library)

- string: character strings
- iostream: input/output, e.g., `cin` and `cout`
- fstream: file input/output, e.g., `ifstream` and `ofstream`
- containers: vector, complex, list, map, ...
- cmath: special functions (inherited from C), e.g. `sqrt`
- cstdlib, cstring, cassert, ...: C header files

Streams

IO

In C++, stream object are responsible for I/O.

You can output an object `obj` to a stream `str` simply by

```
str << obj
```

while you can read an object `obj` from a stream `str` simply by

```
str >> obj
```

The stream will encode these object in ascii format, provided a proper operator is defined (true for the standard c++ types).

Standard streams

- `std::cout` For output to the screen (buffered)
- `std::cin` For input from the keyboard
- `std::cerr` For error messages (by default to the screen too)

These are defined in the header file `iostream`

Streams - File IO

- Classes for file IO are defined in the header `fstream`.
- The `ofstream` class is for output to a file.
- The `ifstream` class is for input from a file.
- You have to declare an object of these classes first.
- Then you can use the streaming operators << and >>.
- Use member functions `read`/`write` to read/write binary.

Streams - File IO

- Classes for file IO are defined in the header `fstream`.
- The `ofstream` class is for output to a file.
- The `ifstream` class is for input from a file.
- You have to declare an object of these classes first.
- Then you can use the streaming operators << and >>.
- Use member functions `read`/`write` to read/write binary.

Example

```
std::ofstream fout("output.txt");
int x = 4;
float y = 1.5;
fout << x << ' ' << y << std::endl;
fout.close();
std::ifstream fin("output.txt");
int x2;
float y2;
fin >> x >> y;
fin.close();
```