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

SciNet HPC Consortium

University of Toronto

Winter 2014

SciNet compute • calcul CANADA

About the course

- Whole-term graduate course
- Prerequisite: basic C, C++ or Fortran experience.
- Will use C++
- Topics include: Scientific computing and programming skills, Parallel programming, and Hybrid programming.

SciNet compute • calcul CANADA

About the course

There are three parts to this course:

1. Scientific Software Development: Jan 2014
   C++, git, make, modular programming, testing, debugging, profiling

2. Numerical Tools for Physical Scientists: Feb/Mar 2014
   modelling, floating point, Random numbers and Monte Carlo, ODE and MD, linear algebra, fft

3. High Performance Scientific Computing: Mar/Apr 2014
   parallel programming, openmp, mpi and hybrid programming

Can be taken separately by astrophysics students as mini-courses, by physics and chemistry students as modular courses.

SciNet
compute • calcul
CANADA

About part 1 of the course

Scientific Software Development

- **Prerequisites**:
  Some programming experience. Some unix prompt experience.
- **Software that you'll need**:
  A unix-like environment with the GNU compiler suite (e.g. Cygwin).
- **Grading scheme**
  Four home work sets, given out on Thursdays.
  To be returned by email by 9:00 am the next Thursday.
- **Course website**

https://support.scinet.utoronto.ca/education/go.php/28/index.php

This site will be kept up-to-date as the course progresses.

SciNet compute • calcul CANADA

About part 1 of the course

Scientific Software Development

- **Instructors**

  Scott Northrup - 256 McCaul Street, Rm 230
  Erik Spence - 256 McCaul Street, Rm 215
  Ramses van Zon - 256 McCaul Street, Rm 228

- **Office Hours**: TBD

- **Location and Times**
  256 McCaul Street, Toronto, ON, Rm 229
  Tuesdays 11:00 am
  Thursdays 11:00 am

- **Dates**
  January 7, 9, 14, 16, 21, 23, 28, and 30, 2014

SciNet
compute • calcul
CANADA

About part 1 of the course

Scientific Software Development

Roadmap

Lecture 1 C++ intro

Lecture 2 More C++, templates

Lecture 3 Modular programming, version control, make

Lecture 4 Object oriented programming

Lecture 5 Multi-dimensional arrays

Lecture 6 Debugging

Lecture 7 Testing and Profiling

Lecture 8 Data management

SciNet compute • calcul CANADA

About part 2 of the course

Numerical Tools for Physical Scientists

- **Prerequisites**:
  Part 1 or solid c++ programming skills, including make and unix/linux prompt experience.
- **Software that you'll need**:
  A unix-like environment with the GNU compiler suite (e.g. Cygwin).
- **Instructors, office hours, grading the same as for part 1**
- **Dates**
  February 4, 6, 11, 13, 25, and 27, 2014 March 4 and 6, 2014

SciNet
compute • calcul
CANADA

About part 2 of the course

Numerical Tools for Physical Scientists

Roadmap

Lecture 9 Modeling, Validation, Verification

Lecture 10 Random numbers and Monte Carlo

Lecture 11 Optimization,root finding

Lecture 12 ODEs and Molecular Dynamics

Lecture 13 Linear Algebra part I

Lecture 14 Linear Algebra part II

Lecture 15 Fast Fourier Transform part I

Lecture 16 Fast Fourier Transform part II

SciNet compute • calcul CANADA

About part 3 of the course

High Performance Scientific Computing

- **Prerequisites**:
  Part 1 or good c++ programming skills, including make and unix/linux prompt experience.

- **Software that you'll need**:
  You will need to bring a laptop with a ssh facility. Hands-on parts will be done on SciNet's GPC cluster.
  For those who don't have a SciNet account yet, the instructions can be found at
  http://wiki.scinethpc.ca/wiki/index.php/Essentials#Accounts

- **Instructors, office hours, grading the same as parts 1,2**

- **Dates**
  March 11, 13, 18, 20, 25, and 27, 2014 April 1 and 3, 2014

SciNet
compute • calcul
CANADA

About part 3 of the course

High Performance Scientific Computing

Roadmap

Lecture 17 Intro to Parallel Computing

Lecture 18 Parallel Computing Paradigms

Lecture 19 Shared Memory Programming with OpenMP, part 1

Lecture 20 Shared Memory Programming with OpenMP part 2

Lecture 21 Distributed Parallel Programming with MPI, part 1

Lecture 22 Distributed Parallel Programming with MPI, part 2

Lecture 23 Distributed Parallel Programming with MPI, part 3

Lecture 24 Hybrid OpenMPI+MPI Programming

SciNet compute • calcul CANADA

Part I

Introduction to Software Development

Lecture 1

Programming

C++ Introduction

- Objects
- Libraries

SciNet compute • calcul CANADA

Programming

- We program to have the computer perform a number of similar computations or data manipulations.
- A program specifies the actions that the computer should take, as well as (restrictions on) the order in which they should be taken.
- Each action will have a net effect on the program's 'state'
- There is limited set of predefined actions, in terms of which we must express all other actions: that is programming.

Programming

- A common pattern of actions to achieve a specific net effect is an algorithm.
- A function is a specification of actions that can be used as a newly defined action.
- A program is a function that can be executed.
- Programs may accept some external data as input and produce data as output.

Program State

- Program state is stored in memory.
- At least part of the state is made up of the program's variables.
- Variables are values that are assigned to a variable name.
- This variable name is associated with a portion of memory that holds the variable's value.

Control structures

- Some actions could be done conditionally on the state of the program and external input. Conditional control structures perform a different actions depending on whether a certain assertion of the state of the system is true.
- Repetition of a set of actions: loops.

Some ideas were taken from:

"A Short Introduction to the Art of Programming" (E. W. Dijkstra, 1971)
http://www.cs.utexas.edu/users/EWD/transcriptions/EWD03xx/EWD316.3.html

Why C++?

Advantages

- High performance
- Low-level programming
- Ubiquitous and standardized
- Useful libraries
- Modular design

Disadvantages

- Labour intensive, error prone
- High-level programming up to you
- Non-interactive
- Things like graphics can be hard
- Beware of performance pitfalls

For e.g. Python, advantages and disadvantages almost reversed.

No free lunch:

You can program poorly and inefficiently in any language.

C++ Introduction

- C/C++ are compiled languages: their basic 'actions' are to be compiled into a set of basic instructions that the processor can execute.
- C was designed for (unix) system programming.
- C has a very small base.
- Most functionality is in (standard) libraries.
- C++ is almost a superset of C.

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream>
int main()
{
   std::string message = "Hello world.";
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream>
int main()
{
   std::string message = "Hello world.";
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream>
int main()
{
   std::string message = "Hello world.";
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main()
{
   std::string message = "Hello world.";
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{
   std::string message = "Hello world.";
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{ // braces delimit a code block
   std::string message = "Hello world."; // a variable
   std::cout << message
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{ // braces delimit a code block
   std::string message = "Hello world."; // a variable
   std::cout << message    // print to standard out
             << std::endl;

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{ // braces delimit a code block
   std::string message = "Hello world."; // a variable
   std::cout << message    // print to standard out
             << std::endl; // end of line

   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{ // braces delimit a code block
   std::string message = "Hello world."; // a variable
   std::cout << message    // print to standard out
             << std::endl; // end of line
   // semicolon after statement
   return 0;

}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

C++ Introduction

Example (Basic C++ program)

```
// hw.cc - prints 'Hello world.'
#include <iostream> // to define input/output routines
int main() // called first
{ // braces delimit a code block
   std::string message = "Hello world."; // a variable
   std::cout << message    // print to standard out
             << std::endl; // end of line
   // semicolon after statement
   return 0;
   // return value to shell
}
```

```
$ g++ -o hw hw.cc [-g -O2]
$ ./hw
Hello world.
$ echo $?
0
```

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

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
```

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: Classes and 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 and objects?

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

```
classname object(arguments);
```

C++ Intro: Classes and objects

Syntax:

```
classname object(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

- In *generic programming*, specific types are not specified initially, but instantiated when needed.
- In C++, generic programming uses templates.
- Many templated functions and classes in the standard library.

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();
```