

Basic syntax in C++

Shahram Rahatlou

Computing Methods in Physics

<http://www.roma1.infn.it/people/rahatlou/cmp/>

Anno Accademico 2020/21



SAPIENZA
UNIVERSITÀ DI ROMA

Brief History of C

- ▷ C was developed in 1967 mainly as a language for writing operating systems and compilers
 - Think about the gcc compiler and Linux today
 - You can compile the gcc compiler yourself
 - You can get the latest linux kernel (core of the linux operating system) from www.kernels.org and compile it yourself
- ▷ C was the evolution of two previous languages: B and BCPL
 - Both used to develop early versions of UNIX at Bell Labs
- ▷ C became very popular and was ported to variety of different hardware platforms
 - C was standardized in 1990 by International Organization for Standardization (ISO) and American National Standards Institute (ANSI)
 - ANSI/ISO 9899: 1990

Object Oriented Programming and Birth of C++

- ▷ By 1970's the difficulties of maintaining very large software projects for companies and businesses had lead to structured programming
 - From Wikipedia:
Structured programming can be seen as a subset or subdiscipline of procedural programming, one of the major programming paradigms. It is most famous for removing or reducing reliance on the GOTO statement (also known as "go to")
- ▷ By late 70's a new programming paradigm was becoming trendy: object orientation
- ▷ In early 1980's Bjarne Stroustrup developed C++ using features from C but adding capabilities for object orientation

What is 'Object Oriented Programming' anyway?

- ▷ Objects are software units modelled after entities in real life
 - Objects are entities with attributes: length, density, elasticity, thermal coefficient, color
 - Objects have a behavior and provide functionalities
 - A door can be opened
 - A car can be driven
 - A harmonic oscillator oscillates
 - A nucleus can decay
 - A planet moves in an orbit
- ▷ Object orientation means writing your program in terms of well defined units (called objects) which have attributes and offer functionalities
 - Program consists in interaction between objects using methods offered by each of them

C++ is not C !

- ▷ Don't be fooled by the name!
- ▷ C++ was developed to overcome limitations of C and improve upon it
 - C++ looks like C but feels very differently
 - C++ shares many basic functionalities but improves upon many of them
 - For example input/output significantly better in C++ than in C
- ▷ C excellent language for structural programming
 - Focused around actions on data structures
 - Provides methods which act on data and create data
- ▷ C++ focused on inter-action between objects
 - Objects are 'smart' data structures: data with behavior!

What You Need to compile your C++ Program?

- ▷ On Linux machines you should have the g++ compiler installed by default
- ▷ On Windows you can use the C++ compiler provided by the free version of Visual Studio
- ▷ On Mac OS, you can install the g++ compiler via XCode, available for free on Mac App Store
- ▷ The easiest way is to use the virtual box available on the course website



Structure of a C++ Program

```
// your first C++ application!  
#include <iostream> // required to perform C++ stream I/O  
  
// function main begins program execution  
int main() {  
    return 0; // indicate that program ended successfully  
} // end function main
```

Precompiler/Preprocessor Directives

What is the preprocessor? What does it do?

```
// your first C++ application!  
#include <iostream> // required to perform C++ stream I/O  
  
// function main begins program execution  
int main() {  
    return 0; // indicate that program ended successfully  
} // end function main
```

`iostream` will be included before compiling this code!

What does the Preprocessor do?

Pre-compile only

```
// Foo.h
class Foo {
public:
    Foo() {};
    Foo(int a) { x_ = a; };

private:
    int x_;
};
```

```
// ExamplePreprocessor.cpp
#include "Foo.h"

int main() {

    return 0;
}
```

```
$ g++ -E ExamplePreprocessor.cpp > prep.cc
$ cat prep.cc
# 1 "ExamplePreprocessor.cpp"
# 1 "<built-in>"
# 1 "<command line>"
# 1 "ExamplePreprocessor.cpp"

# 1 "Foo.h" 1
class Foo {
public:
    Foo() {};
    Foo(int a) { x_ = a; };

private:
    int x_;
};
# 3 "ExamplePreprocessor.cpp" 2

int main() {

    return 0;
}
```

- ▷ Replace user directives with requested source code
 - Foo.h is included in ExamplePreprocessor.cpp

Comments in C++

```
// your first C++ application!  
#include <iostream> // required to perform C++ stream I/O  
  
// function main begins program execution  
int main() {  
    return 0; // indicate that program ended successfully  
  
} // end function main
```

▷ Comments preceded by //

- Can start anywhere in the program either at the **beginning** or right after a statement in the middle of the line

Compiling a C++ application

```
$ g++ -o Welcome Welcome.cpp
```

Name of the binary output

C++ file to compile and link

```
$ ls -l
-r--r--r-- 1 rahatlou None 1379 Apr 18 22:55
Welcome.cpp
-rwxr-xr-x 1 rahatlou None 476600 Apr 18 22:57
Welcome
```

- We will be using the free compiler gcc throughout the examples in this course

Some basic aspects of C++

- ▷ All statements must end with a semi-colon ;
 - Carriage returns are not meaningful and ignored by the compiler
- ▷ Comments are preceded by //
 - Comments can be an entire line or in the middle of the line after a statement
- ▷ Any C++ application must have a **main** method
- ▷ **main** must return an **int**
 - Return value can be used by user/client/environment
 - E.g. to understand if there was an error condition

What about changing a different type of **main**?

```
// VoidMain.cpp
#include <iostream>
using namespace std;

void main() {

    // no return type

} // end function main
```

```
$ g++ -o VoidMain VoidMain.cpp
VoidMain.cpp:6: error: `main' must return `int'
```

- ▷ Compiler requires **main** to return an **int** value!
- ▷ Users must simply must satisfy this requirement
 - If you need a different type there is probably a mistake in your design!

Typical Compilation Errors So Far

```
// BadCode1.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

    int nIterations;

    cout << "How many
            iterations? "; // cannot break in the middle of the string!

    cin >> nIteration; // wrong name! the s at the end missing

    // print message to STDOUT
    cout << "Number of requested iterations: " << nIterations << endl;

    return 0 // ; is missing!

} // end of main
```

```
$ g++ -o BadCode1 BadCode1.cpp
BadCode1.cpp: In function `int main()':
BadCode1.cpp:9: error: missing terminating " character
BadCode1.cpp:10: error: `iterations' undeclared (first use this function)
BadCode1.cpp:10: error: (Each undeclared identifier is reported only once for each function
it appears in.)
BadCode1.cpp:10: error: missing terminating " character
BadCode1.cpp:12: error: `nIteration' undeclared (first use this function)
BadCode1.cpp:12: error: expected `:' before ';' token
BadCode1.cpp:12: error: expected primary-expression before ';' token
BadCode1.cpp:19: error: expected `;' before '}' token
```

Some C reminders

Always initialise your variables!

```
// tinput_bad2.cc
#include <iostream>
using namespace std;
```

```
int main() {
```

```
    int iters;
```

```
    cout << "iters before cin: " << iters << endl;
```

```
    cout << "iterations? ";
```

```
    cin >> iters;
```

```
    cout << "requested " << iters << " iterations" << endl;
```

```
    return 0;
```

```
}
```

Random value since
not initialized!



```
$ g++ -Wall -o tinput_bad2 tinput_bad2.cc
```

```
$ ./tinput_bad2
```

```
iters before cin: 134514841
```

```
iterations? 3
```

```
requested 3 iterations
```

```
$ ./tinput_bad2
```

```
iters before cin: 134514841
```

```
iterations? er
```

```
requested 134514841 iterations
```


Arrays (same as in C)

```
// vect3.cc
#include <iostream>
using namespace std;

int main() {

    float vect[3] = {0.4,1.34,56.156}; // vector of int
    float v2[3];
    float v3[] = { 0.9, -0.1, -0.65}; // array of size 3

    for(int i = 0; i<3; ++i) {
        cout << "i: " << i << "\t"
              << "vect[" << i << "]: " << vect[i] << " \t"
              << "v2[" << i << "]: " << v2[i] << " \t"
              << "v3[" << i << "]: " << v3[i]
              << endl;
    }

    return 0;
}
```

Index of arrays starts from 0 !!

v2[0] is the first elements of array v2 of size 3.

v2[2] is the last element of v2

What happened to v2?

```
$ g++ -o vect3 vect3.cc
$ ./vect3
i: 0    vect[0]: 0.4
i: 1    vect[1]: 1.34
i: 2    vect[2]: 56.156
```

```
v2[0]: 5.34218e+36    v3[0]: 0.9
v2[1]: 2.62884e-42    v3[1]: -0.1
v2[2]: 3.30001e-39    v3[2]: -0.65
```

Arrays and Pointers

- The name of the array is a pointer to the first element of the array

```
// array.cpp
#include <iostream>
using namespace std;

int main() {

    int vect[3] = {1,2,3}; // vector of int
    int v2[3]; //what is the default value?
    int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

    int* d = v3;
    int* c = vect;
    int* e = v2;

    for(int i = 0; i<5; ++i) {
        cout << "i: " << i << ", d = " << d << ", *d: " << *d;
        ++d;
        cout << ", c = " << c << ", *c: " << *c;
        ++c;
        cout << ", e = " << e << ", *e: " << *e << endl;
        ++e;
    }
    return 0;
}
```

What happened to e?

```
$ g++ -o array array.cc
$ ./array
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecd, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eeec, *e: 2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

Another bad example of using arrays

```
// vect2.cc
#include <iostream>
using namespace std;

int main() {

    float vect[3] = {0.4,1.34,56.156}; // vector of int
    float v2[3]; // use default value 0 for each element
    float v3[] = { 0.9, -0.1, -0.65, 1.012, 2.23, -0.67, 2.22 }; // array of size 7

    for(int i = 0; i<5; ++i) {
        cout << "i: " << i << "\t"
              << "vect[" << i << "]: " << vect[i] << " \t"
              << "v2[" << i << "]: " << v2[i] << " \t"
              << "v3[" << i << "]: " << v3[i]
              << endl;
    }

    return 0;
}
```

Accessing out of range component!

```
$ g++ -o vect2 vect2.cc
$ ./vect2
i: 0      vect[0]: 0.4          v2[0]: 5.34218e+36      v3[0]: 0.9
i: 1      vect[1]: 1.34        v2[1]: 2.62884e-42     v3[1]: -0.1
i: 2      vect[2]: 56.156      v2[2]: 3.30001e-39     v3[2]: -0.65
i: 3      vect[3]: 5.60519e-45  v2[3]: 1.57344e+20     v3[3]: 1.012
i: 4      vect[4]: 1.72441e+20  v2[4]: 0.4            v3[4]: 2.23
```

Example of Bad non-initialized Arrays

```
// vect1.cc
#include <iostream>
#include <cmath>

using namespace std;

int main() {

    float vect[3]; // no initialization

    cout << "printing garbage since vector not initialized" << endl;
    for(int i=0; i<3; ++i) {
        cout << "vect[" << i << "] = " << vect[i]
            << endl;
    }

    vect[0] = 1.1;
    vect[1] = 20.132;
    vect[2] = 12.66;

    cout << "print vector after setting values" << endl;
    for(int i=0; i<3; ++i) {
        cout << "vect[" << i << "] =      " << vect[i] << "\t"
            << "sqrt( vect[" << i << "] ) = " << sqrt(vect[i])
            << endl;
    }

    return 0;
}
```

```
$ ./vect1
printing garbage since vector not initialized
vect[0] = 2.62884e-42
vect[1] = NaN
vect[2] = 0
print vector after setting values
vect[0] =      1.1          sqrt( vect[0] ) = 1.04881
vect[1] =     20.132        sqrt( vect[1] ) = 4.48687
vect[2] =     12.66        sqrt( vect[2] ) = 3.55809
```

Control Statements in C++

```
// SimpleIf.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

    if( 1 == 0 ) cout << "1==0" << endl;

    if( 7.2 >= 6.9 ) cout << "7.2 >= 6.9" << endl;

    bool truth = (1 != 0);
    if(truth) cout << "1 != 0" << endl;

    if( ! ( 1.1 >= 1.2 ) ) cout << "1.1 < 1.2" << endl;

    return 0;
} // end of main
```

```
$ g++ -o SimpleIf SimpleIf.cpp
$ ./SimpleIf
7.2 >= 6.9
1 != 0
1.1 < 1.2
```

Declaration and Definition of Variables

```
// SimpleVars.cpp
#include <iostream>
using namespace std;

int main() {

    int samples; // declaration only

    int events = 0; // declaration and assignment

    samples = 123; // assignment

    cout << "How many samples? " ;
    cin >> samples; // assignment via I/O

    cout << "samples: " << samples
         << "\t" // insert a tab in the printout
         << "events: " << events
         << endl;

    return 0;
} // end of main
```

```
$ g++ -o SimpleVars SimpleVars.cpp
$ ./SimpleVars
How many samples? 3
samples: 3         events: 0
```

Loops and iterations in C++

```
int main() { // main begins here
```

```
    int nIterations;  
    cout << "How many iterations? ";  
    cin >> nIterations;
```

```
    int step;  
    cout << "step of iteration? " ;  
    cin >> step;
```

```
    for(int index=0; index < nIterations; index+=step) {  
        cout << "index: " << index << endl;  
    }  
    return 0;  
} // end of main
```

Maximum

Starting value

```
$ g++ -o SimpleLoop SimpleLoop.cpp  
$ ./SimpleLoop  
How many iterations? 7  
step of iteration? 3  
index: 0  
index: 3  
index: 6
```

step

Namespace, Pointers and References, Constants

Introduction to Class

Shahram Rahatlou

Computing Methods in Physics

<http://www.roma1.infn.it/people/rahatlou/cmp/>

Anno Accademico 2019/20



SAPIENZA
UNIVERSITÀ DI ROMA

Output with `iostream`

```
// SimpleIO.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

    // print message to STDOUT
    cout << "Moving baby steps in C++!" << endl;

    return 0;

} // end of main
```

End of line
start a new line!

```
$ g++ -o SimpleIO SimpleIO.cpp
$ ./SimpleIO
Moving baby steps in C++!
```

- ▷ `iostream` provides output capabilities to your program

Input with `iostream`

```
// SimpleInput.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

    int nIterations;

    cout << "How many iterations? ";
    cin >> nIterations;

    // print message to STDOUT
    cout << "Number of requested iterations: " << nIterations << endl;

    return 0;
} // end of main
```

Put content of cin into
variable nIterations

```
$ g++ -o SimpleInput SimpleInput.cpp
$ ./SimpleInput
How many iterations? 7
Number of requested iterations: 7
```

▷ `iostream` provides also input capabilities to your program

Problems with `cin`

```
// tinput_bad.cc
#include <iostream>
using namespace std;

int main() {

    cout << "iterations? ";

    int iters;
    cin >> iters;

    cout << "requested " << iters << " iterations" << endl;

    return 0;
}
```

```
$ g++ -Wall -o tinput_bad tinput_bad.cc
$ ./tinput_bad
iterations? 23
requested 23 iterations
$ ./tinput_bad
iterations? dfed
requested 134514793 iterations
```

Checking `cin` success or failure

```
//tinput.cc
#include <iostream>
using namespace std;

int main() {

    cout << "iterations? ";

    int iters = 0;
    cin >> iters;

    if(cin.fail()) cout << "cin failed!" << endl;

    cout << "requested " << iters << " iterations" << endl;

    return 0;
}
```

Fails if input data doesn't match expected data type

```
$ g++ -Wall -o tinput tinput.cc
$ ./tinput
iterations? 34
requested 34 iterations
$ ./tinput
iterations? sfee
cin failed!
requested 0 iterations
```

Scope of Variables

```
// scope.cc
#include <iostream>
```

```
double f1() {
    double y = 2;
    return y;
}
```

```
int main() {

    double x = 3;
    double z = f1();

    std::cout << "x: " << x << ", z: " << z << ", y: " << y
               << std::endl;
    return 0;
}
```

```
$ g++ -o scope scope.cc
scope.cc: In function `int main()':
scope.cc:16: error: `y' undeclared (first use this function)
scope.cc:16: error: (Each undeclared identifier is reported
only once for each function it appears in.)
```

What is the difference
between `cout` and `std::cout`?

- The scope of a name is the block of program where the name is valid and can be used
 - A block is delimited by `{ }`
 - It can be the body of a method, or a simple scope defined by the user using `{ }`

What is **namespace** ?

- ▷ A mechanism to group declarations that logically belong to each other

```
namespace physics {  
    class vector;  
    class unit;  
    class oscillator;  
    void sort(const vector& value);  
}  
  
namespace electronics {  
    void sort(const vector& value);  
    class oscillator;  
}  
  
namespace graphics {  
    void sort(const vector& value);  
    class unit;  
}
```

- ▷ Provides an easy way for logical separation of parts of a big project
- ▷ Basically a ‘scope’ for a group of related declarations

How do I use namespaces ?

```
#include <iostream>

namespace physics {
    double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
    double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

int main() {
    double x = 3;
    double y = 4;

    double z1 = physics::mean(x,y);
    std::cout << "physics::mean(" << x << "," << y << ") = " << z1
               << std::endl;

    double z2 = foobar::mean(x,y);
    std::cout << "foobar::mean(" << x << "," << y << ") = " << z2
               << std::endl;

    return 0;
}
```

physics::mean

foobar::mean

Use "::" to specify the namespace

```
$ g++ -o namespace1 namespace1.cc
$ ./namespace1
physics::mean(3,4) = 3.5
foobar::mean(3,4) = 12.5
```

Defined in iostream

Common Errors with namespaces

```
// namespaceBad.cc
#include <iostream>

namespace physics {
    double mean(const double& a, const double& b) {
        return (a+b)/2.;
    }
}

int main() {

    double x = 3;
    double y = 4;

    double z1 = mean(x,y); // forgot the namespace!
    cout << "physics::mean(" << x << "," << y << ") = " << z1
        << std::endl;

    return 0;
}
```

If you forget to specify the namespace the compiler doesn't know where to find the method

```
$ g++ -o namespaceBad namespaceBad.cc
namespaceBad.cc: In function `int main()':
namespaceBad.cc:15: error: `mean' undeclared (first use this function)
namespaceBad.cc:15: error: (Each undeclared identifier is reported only
once for each function it appears in.)
namespaceBad.cc:16: error: `cout' undeclared (first use this function)
```


using namespace directive

```
// namespace2.cc
#include <iostream>

namespace physics {
    double mean(const double& a, const double& b) {
        return (a+b)/2.;
    }
}

using namespace std; // make all names in std namespace available!

int main() {

    double x = 3;
    double y = 4;

    double z1 = physics::mean(x,y);
    cout << "physics::mean(" << x << ", " << y << ") = " << z1
         << endl;

    return 0;
}
```

Provide default namespace
for un-qualified names

Same
concepts used
also in python

Compiler looks for `cout` and `endl` first
if not found looks for `std::cout` and
`std::endl`;

```
$ g++ -o namespace2 namespace2.cc
$ ./namespace2.exe
physics::mean(3,4) = 3.5
```

Be careful with `using` directive!

```
// namespaceBad2.cc
#include <iostream>

namespace physics {
    double mean(const double& a, const double& b) { return (a+b)/2.; }
}

namespace foobar {
    double mean(const double& a, const double& b) { return (a*a+b*b)/2.; }
}

using namespace foobar;
using namespace physics;
using namespace std;

int main() {
    double x = 3;
    double y = 4;

    double z1 = mean(x,y);
    double z2 = mean(x,y);

    return 0;
}
```

Ambiguous use of
method `mean`!

Is it in `foobar` or in `physics`?

```
$ g++ -o namespaceBad2 namespaceBad2.cc
namespaceBad2.cc: In function `int main()':
namespaceBad2.cc:21: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:   double foobar::mean(const double&, const double&)
namespaceBad2.cc:25: error: call of overloaded `mean(double&, double&)' is ambiguous
namespaceBad2.cc:5: note: candidates are: double physics::mean(const double&, const double&)
namespaceBad2.cc:9: note:   double foobar::mean(const double&, const double&)
```

Some tips on `using` directive

```
// namespace3.cc
#include <iostream>

namespace physics {
    double mean(const double& a, const double& b) {
        return (a+b)/2.;
    }
}

void printMean(const double& a, const double& b) {
    double z1 = physics::mean(a,b);

    using namespace std; // using std namespace within this method!
    cout << "physics::mean(" << a << ", " << b << ") = " << z1 << endl;
}

int main() {

    double x = 3;
    double y = 4;
    printMean(x,y);

    cout << "no namespace available in the main!" << endl;
    return 0;
}
```

Namespace defined
only within printMean

```
$ g++ -o namespace3 namespace3.cc
namespace3.cc: In function `int main()':
namespace3.cc:23: error: `cout' undeclared (first use this function)
namespace3.cc:23: error: (Each undeclared identifier is reported only
once for each function it appears in.)
namespace3.cc:23: error: `endl' undeclared (first use this function)
```

No default namespace in the main()

Another Example on Scopes

```
#include <iostream>
//using namespace std;

using std::cout;
using std::endl;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    { // just a local scope
        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

Another way to declare
ONLY classes and functions
we are going to use
instead of entire namespace

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: ???
in local scope before int, x: ???
in local scope after int, x: ???
in main after local scope, x: ???
```

What do you think the output
is going to be?

Another Example on Scopes

```
#include <iostream>
//using namespace std;

using std::cout;
using std::endl;

int main() {

    double x = 1.2;

    cout << "in main before scope, x: " << x << endl;

    { // just a local scope
        x++;
        cout << "in local scope before int, x: " << x << endl;

        int x = 4;
        cout << "in local scope after int, x: " << x << endl;
    }

    cout << "in main after local scope, x: " << x << endl;

    return 0;
}
```

Another way to declare
ONLY classes and functions
we are going to use
instead of entire namespace

Changed value of x from main scope

Define new variable in this scope

Back to the main scope

```
$ g++ -o scope scope.cc
$ ./scope
in main before scope, x: 1.2
in local scope before int, x: 2.2
in local scope after int, x: 4
in main after local scope, x: 2.2
```

Functions and Methods

- ▷ A function is a set of operations to be executed
 - Typically there is some input to the function
 - Usually functions have a return value
 - Functions not returning a specific type are **void**

```
// func1.cc
#include <iostream>

double pi() {
    return 3.14;
}

void print() {
    std::cout << "void function print()" << std::endl;
}

int main() {

    std::cout << "pi: " << pi() << std::endl;
    print();

    return 0;
}
```

```
$ g++ -o func1 func1.cc
$ ./func1
pi: 3.14
void function print()
```

Functions must be declared before being used

```
// func2.cc
#include <iostream>

double pi() {
    return 3.14;
}

int main() {

    std::cout << "pi: " << pi() << std::endl;
    print();

    return 0;
}

void print() {
    std::cout << "void function print()" << std::endl;
}
```

Compiler does not know
what the name `print` stands for!

No declaration at this point!

```
$ g++ -o func2 func2.cc
func2.cc: In function `int main()':
func2.cc:11: error: `print' undeclared (first use this function)
func2.cc:11: error: (Each undeclared identifier is reported only
once for each function it appears in.)
func2.cc: In function `void print()':
func2.cc:16: error: `void print()' used prior to declaration
```

Definition can be elsewhere

```
// func3.cc
#include <iostream>

double pi() {
    return 3.14;
}

extern void print(); // declare to compiler print() is a void method

int main() {

    std::cout << "pi: " << pi() << std::endl;
    print();

    return 0;
}

// now implement/define the method void print()
void print() {
    std::cout << "void function print()" << std::endl;
}
```

```
$ g++ -o func3 func3.cc
$ ./func3
pi: 3.14
void function print()
```


Pointers and References

- ▷ A variable is a label assigned to a location of memory and used by the program to access that location

`int a`



4 bytes==32bit of memory

```
// Pointers.cpp
#include <iostream>
using namespace std;

int main() { // main begins here

    int a; // a is a label for a location of memory storing an int value

    cout << "Insert value of a: ";
    cin >> a; // store value provided by user
              // in location of memory held by a

    int* b; // b is a pointer to variable of
            // type a

    b = &a; // value of b is the address of memory
            // location assigned to a

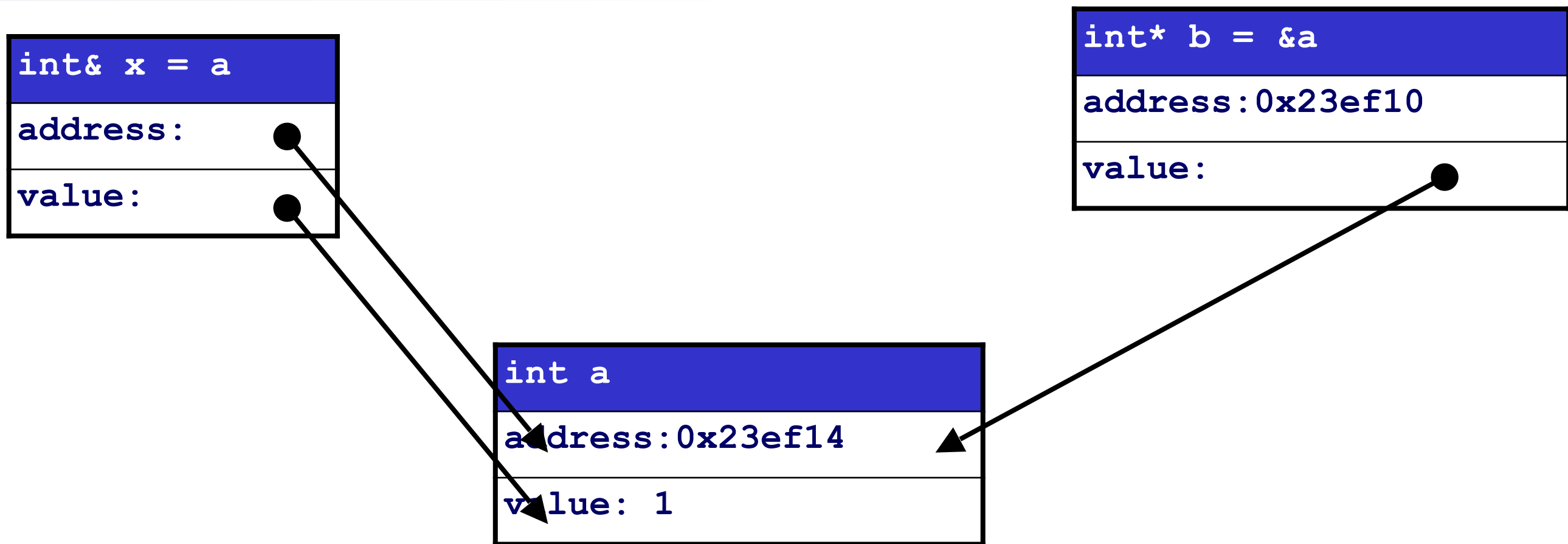
    cout << "value of a: " << a << endl;
    cout << "address of a: " << b << endl;

    return 0;
} // end of main
```

Same location
in memory but
different values!

```
$ g++ -o Pointers Pointers.cpp
$ ./Pointers
Insert value of a: 3
value of a: 3
address of a: 0x23ef14
$ ./Pointers
Insert value of a: 1.2
value of a: 1
address of a: 0x23ef14
```

Pointers and References



- ▷ **x** is a reference to **a**
 - A different name for the same physical location in memory
 - Using **x** or **a** is exactly the same!
- ▷ **b** is a pointer to location of memory named **x** or **a**

Pointers and References

```
// refs.cpp
#include <iostream>
using namespace std;

int main() {

    int a = 1;

    int* b; // b is a pointer to variable of type int

    b = &a; // value of b is the address of memory location assigned to a

    int& x = a; //

    cout << "value of a: " << a
         << ", address of a, &a: " << &a
         << endl;

    cout << "value of x: " << x
         << ", address of x, &x: " << &x
         << endl;

    cout << "value of b: " << b
         << ", address of b, &b: " << &b
         << ", value of *b: " << *b
         << endl;

    return 0;
}
```

```
$ ./refs
value of a: 1, address of a, &a: 0x23ef14
value of x: 1, address of x, &x: 0x23ef14
value of b: 0x23ef14, address of b, &b: 0x23ef10, value of *b: 1
```

Using pointers and references

```
// refs2.cpp
#include <iostream>
using namespace std;

int main() {

    int a = 1;

    int* b = &a;
    *b = 3;

    cout << "value of a: " << a
         << ", address of a, &a: " << &a
         << endl;

    int& x = a;
    x = 45;

    cout << "value of a: " << a
         << ", address of a, &a: " << &a
         << endl;

    return 0;
}
```

Change value of a
with pointer b

Change value of a
with reference x

```
$ g++ -o refs2 refs2.cc
$ ./refs2
value of a: 3, address of a, &a: 0x23ef14
value of a: 45, address of a, &a: 0x23ef14
```

Bad and Null Pointers

```
// badptr1.cpp
#include <iostream>
using namespace std;

int main() {

    int* b; // b is a pointer to variable of type int

    int vect[3] = {1,2,3}; // vector of int

    int* c; // non-initialized pointer
    cout << "c: " << c << ", *c: " << *c << endl;

    for(int i = 0; i<3; ++i) {
        c = &vect[i];
        cout << "c = &vect[" << i << "]: " << c << ", *c: " << *c << endl;
    }

    // bad pointer
    c++;
    cout << "c: " << c << ", *c: " << *c << endl;

    // null pointer causing trouble
    c = 0;
    cout << "c: " << c << endl;
    cout << "*c: " << *c << endl;

    return 0;
}
```

No problem compiling

Crash at runtime

What is the size of an int in memory?

```
$ g++ -o badptr1 badptr1.cc
$ ./badptr1
c: 0x7c90d592, *c: -1879046974
c = &vect[0]: 0x23eef0, *c: 1
c = &vect[1]: 0x23eef4, *c: 2
c = &vect[2]: 0x23eef8, *c: 3
c: 0x23eefc, *c: 1627945305
c: 0
Segmentation fault (core dumped)
```

Constants

- C++ allows to ensure value of a variable does not change within its scope
 - Can be applied to variables, pointers, references, vectors etc.
 - Constants must be ALWAYS initialized since they can't change at a later time!

```
// const1.cpp

int main() {

    const int a = 1;
    a = 2;

    const double x;

    return 0;
}
```

```
$ g++ -o const1 const1.cc
const1.cc: In function `int main()':
const1.cc:6: error: assignment of read-only variable `a'
const1.cc:8: error: uninitialized const `x'
```

Constant Pointer

Read from right to left:

`int * const b:`

`b` is a constant pointer to `int`

```
// const2.cpp
```

```
int main() {
```

```
    int a = 1;
```

```
    int * const b = &a; // const pointer to int
```

```
    *b = 5; // OK. can change value of what b points to
```

```
    int c = 3;
```

```
    b = &c; // Not OK. assign new value to c
```

```
    return 0;
```

```
}
```

```
$ g++ -o const2 const2.cc
```

```
const2.cc: In function `int main()':
```

```
const2.cc:11: error: assignment of read-only variable `b'
```

Pointer to Constant

a is not a constant!

But we can treat it as such when pointing to it

```
// const3.cpp

int main() {

    int a = 1;
    const int * b = &a; // pointer to const int

    int c = 3;
    b = &c; // assign new value to c ... OK!

    *b = 5; // assign new value to what c point to ... NOT OK!

    return 0;
}
```

Read from right to left:

`const int * b:`

b is a pointer to constant int

```
$ g++ -o const3 const3.cc
const3.cc: In function `int main()':
const3.cc:11: error: assignment of read-only location
```

**NB: the error
is different!**

Constant Pointer to Constant Object

- Most restrictive access to another variable
 - Specially when used in function interface
- Can not change neither the pointer nor what it points to!

```
// const4.cpp

int main() {

    float a = 1;
    const float * const b = &a; // const pointer to const float

    *b = 5; // Not OK. can't change value of what b points to

    float c = 3;
    b = &c; // Not OK. can't change what b points to!

    return 0;
}
```

```
$ g++ -o const4 const4.cc
const4.cc: In function `int main()':
const4.cc:8: error: assignment of read-only location
const4.cc:11: error: assignment of read-only variable
```

Bad Use of Pointers

```
int vect[3] = {1,2,3};
int v2[3];
int v3[] = { 1, 2, 3, 4, 5, 6, 7 };
```

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

```
$ ./array
```

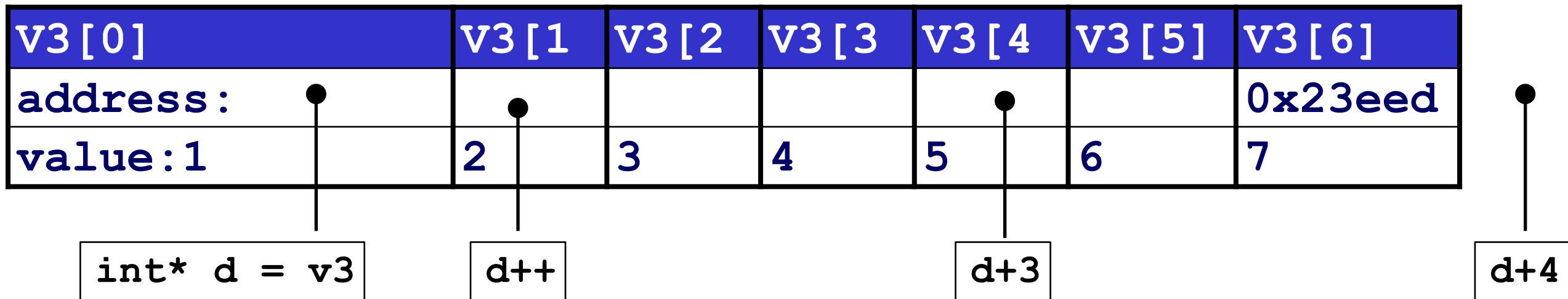
```
i: 0, d = 0x23eec0, *d: 1, c = 0x23eef0, *c: 1, e = 0x23eee0, *e: -1
i: 1, d = 0x23eec4, *d: 2, c = 0x23eef4, *c: 2, e = 0x23eee4, *e: 2088773120
i: 2, d = 0x23eec8, *d: 3, c = 0x23eef8, *c: 3, e = 0x23eee8, *e: 2088772930
i: 3, d = 0x23eecd, *d: 4, c = 0x23eefc, *c: 1627945305, e = 0x23eecd, *e:
2089866642
i: 4, d = 0x23eed0, *d: 5, c = 0x23ef00, *c: 1876, e = 0x23eef0, *e: 1
```

v3[0]	v3[1]	v3[2]	v3[3]	v3[4]	v3[5]	v3[6]
address:						0x23eed
value:1	2	3	4	5	6	7

How many bytes in memory between v3[6] and v2[0] ?

v2[0]	v2[1]	v2[2]		vect[0]	vect[1]	Vect[2]
0x23eee	0x23eee	0x23eee	0x23eee	0x23eef	0x23eef	0x23eef
-1	2	3		1	2	3

Pointer Arithmetic



```
// ptr.cc
#include <iostream>
using namespace std;

int main() {

    int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7
    int* d = v3;
    cout << "d = " << d << ", *d: " << *d << endl;

    d++;
    cout << "d = " << d << ", *d: " << *d << endl;

    d = d+3;
    cout << "d = " << d << ", *d: " << *d << endl;

    d = d+4;
    cout << "d = " << d << ", *d: " << *d << endl;

    return 0;
}
```

```
$ g++ -o ptr ptr.cc
$ ./ptr
d = 0x23eef0, *d: 1
d = 0x23eef4, *d: 2
d = 0x23ef00, *d: 5
d = 0x23ef10, *d: 1628803505
```

+ and - operators with Pointers

```
// ptr2.cc
#include <iostream>
using namespace std;

int main() {

    int v3[] = { 1, 2, 3, 4, 5, 6, 7 }; // array of size 7

    int* d = v3;
    int* c = &v3[4];
    cout << "d = " << d << ", *d: " << *d << endl;
    cout << "c = " << c << ", *c: " << *c << endl;

    //int* e = c + d; // not allowed

    cout << "c-d: " << c - d << endl;
    cout << "d-c: " << d - c << endl;

    //int* e = c-d; // wrong!

    int f = c - d;
    float g = c - d;

    cout << "f: " << f << " g: " << g << endl;

    int * h = &v3[6] + (d-c);
    cout << "int * h = &v3[6] + (d-c): " << h << " *h: " << *h << endl;

    return 0;
}
```

Arguments of Functions

- Arguments of functions can be passed in two different ways

```
// funcarg1.cc
#include <iostream>

using namespace std;

void emptyLine() {
    cout
    << "\n-----\n"
    << endl;
}
```

– By value

- x is a local variable in f1()

```
void f1(double x) {
    cout << "f1: input value of x = "
    << x << endl;
    x = 1.234;
    cout << "f1: change value of x in f1(). x = "
    << x << endl;
}
```

– Pointer or reference

- x is reference to argument used by caller

```
void f2(double& x) {
    cout << "f2: input value of x = "
    << x << endl;
    x = 1.234;
    cout << "f2: change value of x in f2(). x = "
    << x << endl;
}
```

Pointers and References in Functions

```
int main() {  
  
    double a = 1.; // define a  
  
    emptyLine();  
    cout << "main: before calling f1, a = " << a << endl;  
    f1(a); // void function  
    cout << "main: after calling f1, a = " << a << endl;  
  
    emptyLine();  
    cout << "main: before calling f2, a = " << a << endl;  
    f2(a); // void function  
    cout << "main: after calling f2, a = " << a << endl;  
  
    return 0;  
}
```

f1 has no effect on
variables in main

Because a is passed by
value

x is a copy of a

f2 modifies the value of
the variable in the main!

Because a is passed by reference

```
double& x = a;
```

```
$ ./funcarg1
```

```
-----
```

```
main: before calling f1, a = 1
```

```
f1: input value of x = 1
```

```
f1: change value of x in f1(). x = 1.234
```

```
main: after calling f1, a = 1
```

```
-----
```

```
main: before calling f2, a = 1
```

```
f2: input value of x = 1
```

```
f2: change value of x in f2(). x = 1.234
```

```
main: after calling f2, a = 1.234
```

Constant Pointers and References in Functions

```
// funcarg2.cc
#include <iostream>

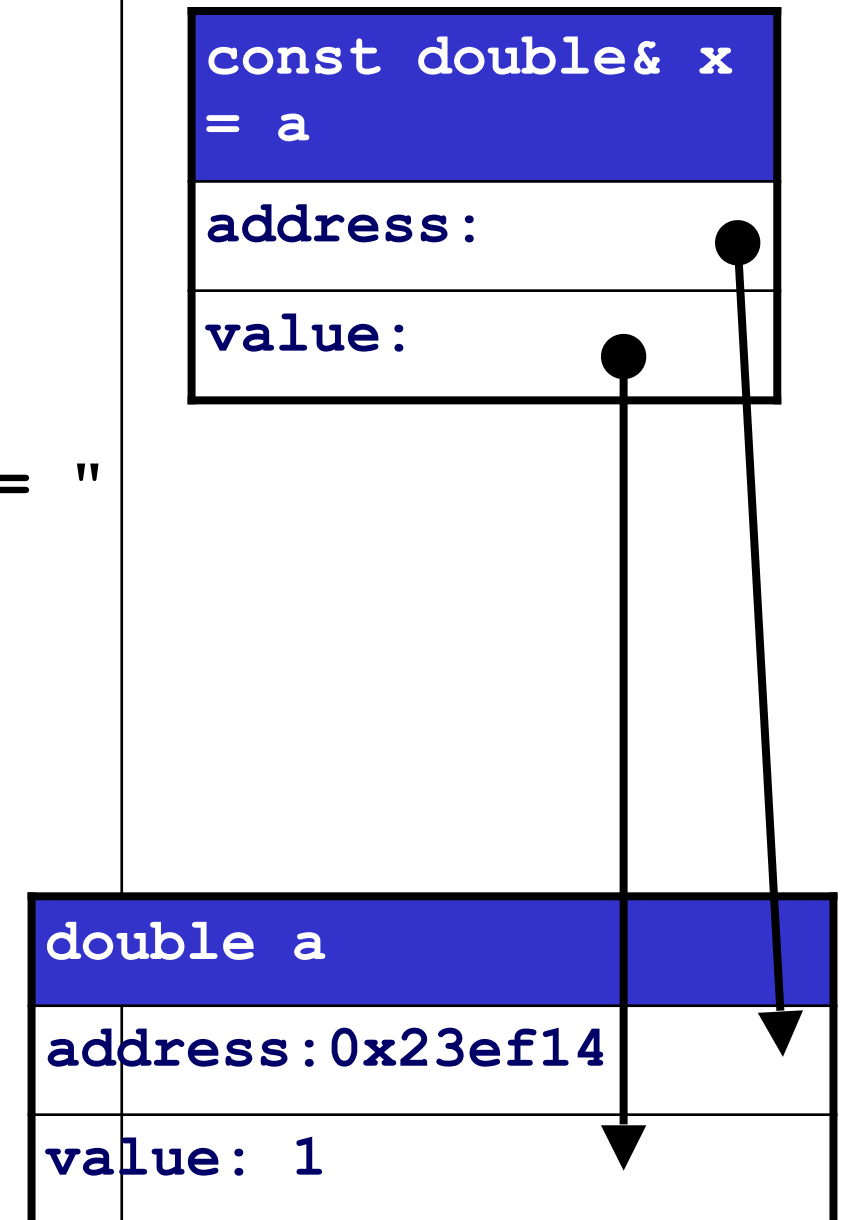
using namespace std;

void f2(const double& x) {
    cout << "f2: input value of x = "
          << x << endl;
    x = 1.234;
    cout << "f2: change value of x in f2(). x = "
          << x << endl;
}

int main() {

    double a = 1.;
    f2(a);

    return 0;
}
```



```
$ g++ -o funcarg2 funcarg2.cc
funcarg2.cc: In function `void f2(const double&)':
funcarg2.cc:9: error: assignment of read-only reference `x'
```

Pointers, References and Passing by Value in Functions

```
// mean.cc
#include <iostream>
using namespace std;

void computeMean(const double* data, int nData, double& mean) {
    mean = 0.;
    for(int i=0; i<nData; ++i) {
        cout << "data: " << data << ", *data: " << *data << endl;
        mean += *data;
        data++;
    }
    mean /= nData; // divide by number of data points
}

int main() {

    double pressure[] = { 1.2, 0.9, 1.34, 1.67, 0.87, 1.04, 0.76 };
    double average;

    computeMean( pressure, 7, average );

    cout << "average pressure: "
         << average << endl;
    return 0;
}
```

```
$ g++ -o mean mean.cc
$ ./mean
data: 0x23eed0, *data: 1.2
data: 0x23eed8, *data: 0.9
data: 0x23eee0, *data: 1.34
data: 0x23eee8, *data: 1.67
data: 0x23eef0, *data: 0.87
data: 0x23eef8, *data: 1.04
data: 0x23ef00, *data: 0.76
average pressure: 1.11143
```


Closer Look at `computeMean()`

```
void computeMean(const double* data, int nData, double& mean) {  
    mean = 0.;  
    for(int i=0; i<nData; ++i) {  
        cout << "data: " << data << ", *data: " << *data << endl;  
        mean += *data;  
        data++;  
    }  
    mean /= nData; // divide by number of data points  
}
```

- Input data passed as constant pointer
 - Good: can't cause trouble to caller! Integrity of data guaranteed
 - Bad: No idea how many data points we have!
- Number of data pointer passed by value
 - Simple int. No gain in passing by reference
 - Bad: separate variable from array of data. Exposed to user error
- Very bad: void function with no return type
 - Good: appropriate name. `computeMean()` suggests an action not a type

New implementation with Return Type

```
double mean(const double* data, int nData) {  
    double mean = 0.;  
    for(int i=0; i<nData; ++i) {  
        cout << "data: " << data << ", *data: " << *data << endl;  
        mean += *data;  
        data++;  
    }  
    mean /= nData; // divide by number of data points  
    return mean  
}
```

- Make function return the computed mean
- New name to make it explicit function returns something
 - Not a rule, but simple courtesy to users of your code
- No need for variables passed by reference to be modified in the function
- Still exposed to user error...

Possible Problems with use of Pointers

```
// mean2.cc
#include <iostream>
using namespace std;

double mean(const double* data, int nData) {

    double mean = 0.;
    for(int i=0; i<nData; ++i) {
        cout << "data: " << data << ", *data: " << *data << endl;
        mean += *data;
        data++;
    }
    mean /= nData; // divide by number of data points
    return mean;
}

int main() {
    double pressure[] = { 1.2, 0.6, 1.8 }; // only 3 elements
    double average = mean(pressure, 4); // mistake!
    cout << "average pressure: " << average << endl;

    return 0;
}
```

```
$ g++ -o mean2 mean2.cc
$ ./mean2
data: 0x23eef0, *data: 1.2
data: 0x23eef8, *data: 0.6
data: 0x23ef00, *data: 1.8
data: 0x23ef08, *data: 8.48798e-314
average pressure: 0.9
```

Simple luck!
Additional value
not changing the
average!

No protection against
possible errors!

What about computing other quantities?

- What if we wanted to compute also the standard deviation of our data points?

```
void computeMean(const double* data, int nData, double& mean, double& stdDev) {  
    // two variables passed by reference to void function  
    // not great. But not harmful.  
}  
  
double meanWithStdDev(const double* data, int nData, double& stdDev) {  
    // error passed by reference to mean function! ugly!! anti-intuitive  
}  
  
double mean(const double* data, int nData) {  
    // one method to compute only average  
}  
  
double stdDev(const double* data, int nData) {  
    // one method to compute standard deviation  
    // use mean() to compute average needed by std deviation  
}
```

What if we had a new C++ type?

▷ Imagine we had a new C++ type called `Result` including data about both mean and standard deviation

▷ We could then simply do the following

```
Result mean(const double* data, int nData) {  
    Result result;  
    // do your calculation  
    return result;  
}
```

▷ *This is exactly the idea of classes in C++!*

Classes in C++

- ▷ A class is a set of data and functions that define the characteristics and behavior of an object
 - Characteristics also known as attributes
 - Behavior is what an object can do and is referred to also as its interface

Interface
or
Member Functions

```
class Result {  
    public:  
  
    // constructors  
    Result() { }  
    Result(const double& mean, const double& stdDev) {  
        mean_ = mean;  
        stdDev_ = stdDev;  
    }
```

```
    // accessors  
    double getMean() { return mean_; };  
    double getStdDev() { return stdDev_; };
```

Data members or
attributes

```
private:  
    double mean_;  
    double stdDev_;
```

```
};
```

Don't's forget ; at the end of definition!

Using class Result

```
#include <iostream>
using namespace std;

class Result {
public:

    // constructors
    Result() { };
    Result(const double& mean, const double& stdDev) {
        mean_ = mean;
        stdDev_ = stdDev;
    }

    // accessors
    double getMean() { return mean_; };
    double getStdDev() { return stdDev_; };

private:
    double mean_;
    double stdDev_;
};
```

```
int main() {

    Result r1;
    cout << "r1, mean: " << r1.getMean()
          << ", stdDev: " << r1.getStdDev()
          << endl;

    Result r2(1.1, 0.234);
    cout << "r2, mean: " << r2.getMean()
          << ", stdDev: " << r2.getStdDev()
          << endl;

    return 0;
}
```

```
$ g++ -o results2 result2.cc
$ ./results2
r1, mean: NaN, stdDev: 8.48798e-314
r2, mean: 1.1, stdDev: 0.234
```

r1 is ill-defined. Why?

What is wrong with Result::Result() ?

C++ Data Types

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2 or 4 bytes	1 wide character

- Size is architecture dependent!
 - Difference between 32-bit and 64-bit machines
 - Above table refers to typical 32-bit architecture
- `int` is usually has size of 'one word' on a given architecture
- Four integer types: `char`, `short`, `int`, and `long`
 - Each type is at least as large as previous one
 - `size(char) <= size(short) <= size(int) <= size(long)`
- Long `int == int`; similarly `short int == short`

Size of Objects/Types in C++

```
// cpptypes.cc
#include <iostream>
using namespace std;

int main() {
    char*      aChar = "c"; // char
    bool       aBool = true; // boolean
    short      aShort = 33; // short
    long       aLong  = 123421; // long
    int        aInt   = 27; // integer
    float      aFloat = 1.043; // single precision
    double     aDbl   = 1.243e-234; // double precision
    long double aLD   = 0.432e245; // double precision

    cout << "char* aChar = " << aChar << "\tsizeof(" << "*char" << "): " << sizeof(*aChar) << endl;
    cout << "bool aBool = " << aBool << "\tsizeof(" << "bool" << "): " << sizeof(aBool) << endl;
    cout << "short aShort = " << aShort << "\tsizeof(" << "short" << "): " << sizeof(aShort) << endl;
    cout << "long aLong = " << aLong << "\tsizeof(" << "long" << "): " << sizeof(aLong) << endl;
    cout << "int aInt = " << aInt << "\tsizeof(" << "int" << "): " << sizeof(aInt) << endl;
    cout << "float aFloat = " << aFloat << "\tsizeof(" << "float" << "): " << sizeof(aFloat) << endl;
    cout << "double aDbl = " << aDbl << "\tsizeof(" << "double" << "): " << sizeof(aDbl) << endl;
    cout << "long double aLD = " << aLD << "\tsizeof(" << "long double" << "): " << sizeof(aLD) << endl;

    return 0;
}
```

```
$ g++ -o cpptypes cpptypes.cc
$ ./cpptypes
char* aChar = c                sizeof(*char): 1
bool aBool = 1                 sizeof(bool): 1
short aShort = 33              sizeof(short): 2
long aLong = 123421            sizeof(long): 4
int aInt = 27                  sizeof(int): 4
float aFloat = 1.043           sizeof(float): 4
double aDbl = 1.243e-234       sizeof(double): 8
long double aLD = 4.32e+244     sizeof(long double): 12
```

Topics

- **Classes**

- data members and member functions

- **Constructors**

- Special member functions

- **private and public members**

- **Helper and utility methods**

- setters
 - getters
 - accessors

Classes in C++

- A class is a set of data and functions that define the characteristics and behavior of an object
 - Characteristics also known as attributes
 - Behavior is what an object can do and is referred to also as its interface

Interface
or
Member Functions

Data members or
attributes

```
class Result {  
    public:  
  
    // constructors  
    Result() { }  
    Result(const double& mean, const double& stdDev) {  
        mean_ = mean;  
        stdDev_ = stdDev;  
    }  
  
    // accessors  
    double getMean() { return mean_; };  
    double getStdDev() { return stdDev_; };  
  
    private:  
    double mean_;  
    double stdDev_;  
};
```

Don't's forget ; at the end of definition!

Data Members (Attributes)

```
class Datum {  
    double value_;  
    double error_;  
};
```

- Data defined in the scope of a class are called data members of that class
- Data members are defined in the class and can be used by all member functions
- Contain the actual data that characterise the content of the class
- Can be public or private
 - public data members are generally bad and symptom of bad design
 - More on this topic later in the course

Interface: Member Functions

- Member functions are methods defined inside the scope of a class
 - Have access to all data members

`name_` is a datamember

No declaration of `name_` in member functions!

`name` is a local variable only within `setName()`

```
// Student
#include <iostream>
#include <string>

class Student {
    using namespace std;
public:
    // default constructor
    Student() { name_ = ""; }

    // another constructor
    Student(const string& name) { name_ = name; }

    // getter method: access to info from the class
    string name() { return name_; }

    // setter: set attribute of object
    void setName(const string& name) { name_ = name; }

    // utility method
    void print() {
        cout << "My name is: " << name_ << endl;
    }

private:
    string name_; // data member
};
```

Arguments of Member Functions

- All C++ rules discussed so far hold
- You can pass variables by value, pointer, or reference
- You can use the constant qualifier to protect input data and restrict the capabilities of the methods
 - This has implications on declaration of methods using constants
 - We will discuss constant methods and data members next week
- Member functions can return any type
 - Exceptions! Constructors and Destructor
 - Have no return type
 - More on this later

Access specifiers **public** and **private**

- Public functions and data members are available to anyone
- Private members and methods are available ONLY to other member functions

```
1 #include <iostream>
2 using std::cout;
3 using std::endl;
4
5 class Datum {
6     public:
7         Datum() { }
8         Datum(double val, double error) {
9             value_ = val;
10            error_ = error;
11        }
12
13        double value() { return value_; }
14        double error() { return error_; }
15
16        void setValue(double value) { value_ = value; }
17        void setError(double error) { error_ = error; }
18
19        double value_; // public data member!!!
20
21    private:
22        double error_; // private data member
23 };
```

Access elements of an object through member selection operator "."

```
25 int main() {
26
27     Datum d1(1.1223,0.23);
28
29     cout << "d1.value(): " << d1.value() ^
30         << " d1.error(): " << d1.error()
31         << endl;
32
33
34     cout << "d1.value_: " << d1.value_
35         << " d1.error_: " << d1.error_
36         << endl;
37
38     return 0;
39 }
```

Accessing private members
is a compilation error!

```
$ g++ -o class1 class1.cc
class1.cc: In function `int main()':
class1.cc:22: error: `double Datum::error_' is private
class1.cc:35: error: within this context
```

private members

```
#include <iostream>
using namespace std;

class Datum {
public:
    Datum(double val, double error) {
        value_ = val;
        error_ = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value)
    { value_ = value; }
    void setError(double error)
    { error_ = error; }

    void print() {
        cout << "datum: " << value_
              << " +/- " << error_
              << endl;
    }

private:
    double value_; // private data member!!!
    double error_; // private data member
};
```

```
int main() {

    Datum d1(1.1223,0.23);
    // setter with no return value
    d1.setValue( 8.563 );

    // getter to access private data
    double x = d1.value();

    cout << "d1.value(): " << d1.value()
          << " d1.error(): " << d1.error()
          << endl;

    d1.print();

    return 0;
}
```

```
$ g++ -o class2 class2.cc
$ ./class2
d1.value(): 8.563 d1.error(): 0.23
datum: 8.563 +/- 0.23
```


private methods

- Can be used only inside other methods but not from outside

```
1 // class3.cc
2 #include <iostream>
3 using namespace std;
4
5 class Datum {
6     public:
7         Datum() { reset(); } // reset data members
8
9         double value() { return value_; }
10        double error() { return error_; }
11
12        void setValue(double value) { value_ = value; }
13        void setError(double error) { error_ = error; }
14
15        void print() {
16            cout << "datum: " << value_ << " +/- "
17                << error_ << endl;
18        }
19    private:
20        void reset() {
21            value_ = 0.0;
22            error_ = 0.0;
23        }
24
25        double value_;
26        double error_;
27 };
```

```
int main() {
    Datum d1;
    d1.setValue( 8.563 );
    d1.print();
    return 0;
}
```

```
$ g++ -o class3 class3.cc
$ ./class3
datum: 8.563 +/- 0
```

```
30 int main() {
31
32     Datum d1;
33     d1.setValue( 8.563 );
34     d1.print();
35     d1.reset();
36
37     return 0;
38 }
```

```
$ g++ -o class4 class4.cc
class4.cc: In function `int main()':
class4.cc:20: error: `void Datum::reset()' is private
class4.cc:35: error: within this context
```

Hiding Implementation from Users/Clients

- How to decide what to make public or private?
- Principle of Least Privilege
 - elements of a class, data or functions, must be private unless proven to be needed as public!
- Users should rely solely on the interface of a class
- They should never use the internal details of the class
- ***That's why having public data members is a VERY bad idea!***
 - name and characteristics of data members can change
 - Functionalities and methods remain the same
 - You must be able to change internal structure of the class without affecting the clients!

Bad Example of Public Data Members

```
class Datum {
public:
    Datum(double val, double error) {
        value_ = val;
        error_ = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
        cout << "datum: " << value_ << " +/- " << error_ << endl;
    }

//private:          // all data are public!
    double value_;
    double error_;
};
```

```
int main() {

    Datum d1(1.1223,0.23);
    double x = d1.value();
    double y = d1.error_;
    cout << "x: " << x << "\t y: " << y << endl;

    return 0;
}
```

application uses directly
the data member!

```
$ g++ -o class6 class6.cc
$ ./class6
x: 1.1223          y: 0.23
```

Bad Example of Public Data Members

Same Application as before

Change the names of data members

No change of functionality so no one should be affected!

```
class Datum {
public:
    Datum(double val, double error) {
        val_ = val;
        err_ = error;
    }

    double value() { return val_; }
    double error() { return err_; }

    void setValue(double value) { val_ = value; }
    void setError(double error) { err_ = error; }

    void print() {
        cout << "datum: " << val_ << " +/- " << err_ << endl;
    }

//private:          // alla data are public!
    double val_;    // value_ → val_
    double err_;    // error_ → err_
};
```

```
28 int main() {
29
30     Datum d1(1.1223,0.23);
31     double x = d1.value();
32     double y = d1.error_;
33
34     cout << "x: " << x << "\t y: " << y << endl;
35
36     return 0;
37 }
```

Our application is now broken!

But Datum has not changed its behavior!

Bad programming!

Only use the interface of an object not its internal data!

Private data members prevent this

```
$ g++ -o class7 class7.cc
class7.cc: In function `int main()':
class7.cc:32: error: 'class Datum' has no member named 'error_'
```

Constructors

```
class Datum {  
    public:  
        Datum() { }  
        Datum(double val, double error) {  
            value_ = val;  
            error_ = error;  
        }  
  
    private:  
        double value_; // public data member!!!  
        double error_; // private data member  
};
```

- Special member functions
 - Required by C++ to create a new object
 - MUST have the same name of the class
 - Used to initialize data members of an instance of the class
 - Can accept any number of arguments
 - Same rules as any other C++ function applies
- Constructors have no return type!
- There can be several constructors for a class
 - Different ways to declare and an object of a given type

Different Types of Constructors

- Default constructor
 - Has no argument
 - On most machines the default values for data members are assigned
- Copy Constructor
 - Make a new object from an existing one
- Regular constructor
 - Provide sufficient arguments to initialize data members

```
class Datum {  
    public:  
        Datum() { }  
  
        Datum(double x, double y) {  
            value_ = x;  
            error_ = y;  
        }  
  
        Datum(const Datum& datum) {  
            value_ = datum.value_;  
            error_ = datum.error_;  
        }  
  
    private:  
        double value_;  
        double error_;  
};
```

Using Constructors

```
// class5.cc
#include <iostream>
using namespace std;

class Datum {
public:
    Datum() { }

    Datum(double x, double y) {
        value_ = x;
        error_ = y;
    }

    Datum(const Datum& datum) {
        value_ = datum.value_;
        error_ = datum.error_;
    }

    void print() {
        cout << "datum: " << value_
              << " +/- " << error_
              << endl;
    }

private:
    double value_;
    double error_;
};
```

```
int main() {

    Datum d1;
    d1.print();

    Datum d2(0.23,0.212);
    d2.print();

    Datum d3( d2 );
    d3.print();

    return 0;
}
```

```
$ g++ -o class5 class5.cc
$ ./class5
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Default Constructors on Different Architectures

```
$ uname -a
CYGWIN_NT-5.1 lajolla 1.5.18(0.132/4/2) 2005-07-02 20:30 i686 unknown
unknown Cygwin
$ gcc -v
Reading specs from /usr/lib/gcc/i686-pc-cygwin/3.4.4/specs
...
gcc version 3.4.4 (cygming special) (gdc 0.12, using dmd 0.125)

$ g++ -o class5 class5.cc
$ ./class5
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Windows XP with CygWin

```
$ uname -a
Linux pccms02.roma1.infn.it 2.6.14-1.1656_FC4smp #1 SMP Thu Jan 5 22:24:06 EST
2006 i686 i686 i386 GNU/Linux
$ gcc -v
Using built-in specs.
Target: i386-redhat-linux
...
gcc version 4.0.2 20051125 (Red Hat 4.0.2-8)
$ g++ -o class5 class5.cc
$ ./class5
datum: 6.3275e-308 +/- 4.85825e-270
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Fedora Core4

Default Assignment

```
// ctor.cc
#include <iostream>
using std::cout;
using std::endl;

class Datum {
public:
    Datum(double x) { x_ = x; }
    double value() { return x_; }
    void setValue(double x) { x_ = x; }
    void print() {
        cout << "x: " << x_ << endl;
    }

private:
    double x_;
};
```

```
int main() {

    Datum d1(1.2);
    d1.print();

    // no default ctor. compiler error if uncommented
    //Datum d2;
    //d2.print();

    Datum d3 = d1; // default assignment by compiler
    d3.print();
    cout << "&d1: " << &d1
         << "\t &d3: " << &d3 << endl;
    return 0;
}
```

d3.x_ = d1.x_
done by compiler

```
$ g++ -o ctor ctor.cc
$ ./ctor
x: 1.2
x: 1.2
&d1: 0x23ef10      &d3: 0x23ef08
```

Question

- Can a constructor be private?
 - Is it allowed by the compiler?
 - How to instantiate an object with no public constructor?

- *Find a working example of a very simple class for next week*

Accessors and Helper/Utility Methods

- Methods that allow read access to data members
- Can also provide functionalities commonly needed by users to elaborate information from the class
 - for example formatted printing of data
- Usually they do not modify the objects, i.e. do not change the value of its attributes

```
class Student {  
    public:  
  
    // getter method: access to data members  
    string name() { return name_; }  
  
    // utility method  
    void print() {  
        cout << "My name is: " << name_ << endl;  
    }  
  
    private:  
    string name_; // data member  
};
```

```
class Datum {  
    public:  
  
    double value() { return value_; }  
    double error() { return error_; }  
  
    void print() {  
        cout << "datum: " << value_  
            << " +/- " << error_  
            << endl;  
    }  
  
    private:  
    double value_; // public data member!!!  
    double error_; // private data member  
};
```

Getter Methods

- getters are helpers methods with explicit names returning individual data members
 - Do not modify the data members simply return them
 - Good practice: call these methods as getFoo() or foo() for member foo_
- Return value of a getter method should be that of the data member

```
class Datum {
public:
    Datum(double val, double error) {
        val_ = val;
        err_ = error;
    }

    double value() { return val_; }
    double error() { return err_; }

    void setValue(double value) { val_ = value; }
    void setError(double error) { err_ = error; }

    void print() {
        cout << "datum: " << val_ << " +/- " << err_
            << endl;
    }

private:
    double val_;
    double err_;
};
```

```
// Student
#include <iostream>
#include <string>
using namespace std;

class Student {
public:
    // default constructor
    Student() { name_ = ""; }

    // another constructor
    Student(const string& name) { name_ = name; }

    // getter method: access to info from the class
    string name() { return name_; }

    // setter: set attribute of object
    void setName(const string& name) { name_ = name; }

    // utility method
    void print() {
        cout << "My name is: " << name_ << endl;
    }

private:
    string name_; // data member
};
```

Setter Methods

- Setters are member functions that modify attributes of an object after it is created
 - Typically defined as void
 - Could return other values for error handling purposes
 - Very useful to assign correct attributes to an object in algorithms
 - As usual abusing setter methods can cause unexpected problems

```
// class8.cc
#include <iostream>
using namespace std;

class Datum {
public:
    Datum(double val, double error) {
        value_ = val;
        error_ = error;
    }

    double value() { return value_; }
    double error() { return error_; }

    void setValue(double value) { value_ = value; }
    void setError(double error) { error_ = error; }

    void print() {
        cout << "datum: " << value_ << " +/- "
              << error_ << endl;
    }

private:
    double value_;
    double error_;
};
```

```
int main() {

    Datum d1(23.4,7.5);
    d1.print();

    d1.setValue( 8.563 );
    d1.setError( 0.45 );
    d1.print();

    return 0;
}
```

```
$ g++ -o class8 class8.cc
$ ./class8
datum: 23.4 +/- 7.5
datum: 8.563 +/- 0.45
```

Pointers and References to Objects

```
// app2.cpp
#include <iostream>
using std::cout; // use using only for specific
classes
using std::endl; // not for entire namespace

class Counter {
public:
    Counter() { count_ = 0; x_=0.0; };
    int value() { return count_; }
    void reset() { count_ = 0; x_=0.0; }
    void increment() { count_++; }
    void increment(int step)
        { count_ = count_+step; }
    void print() {
        cout << "---- Counter::print() ----" << endl;
        cout << "my count_: " << count_ << endl;
        // this is special pointer
        cout << "my address: " << this << endl;
        cout << "&x_ : " << &x_ << " sizeof(x_): "
            << sizeof(x_) << endl;
        cout << "&count_ : " << &count_
            << " sizeof(count_): "
            << sizeof(count_) << endl;
        cout << "---- Counter::print() ----" << endl;
    }

private:
    int count_;
    double x_; // dummy variable
};
```

```
void printCounter(Counter& counter) {
    cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter) {
    cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
    Counter counter;
    counter.increment(7);

    // ptr is a pointer to a Counter Object
    Counter* ptr = &counter;
    cout << "ptr = &counter: " << &counter << endl;

    // use . to access member of objects
    cout << "counter.value(): " << counter.value() << endl;

    // use -> with pointer to objects
    cout << "ptr->value(): " << ptr->value() << endl;

    printCounter( counter );
    printByPtr( ptr );

    ptr->print();

    cout << "sizeof(ptr): " << sizeof(ptr) << "\t"
        << "sizeof(counter): " << sizeof(counter)
        << endl;

    return 0;
}
```

-> instead of . When using pointers to objects

Size and Address of Objects

gcc 3.4.4 on cygwin

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0x22ccd0
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0x22ccd0
&count_ : 0x22ccd0 sizeof(count_): 4
&x_ : 0x22ccd8 sizeof(x_): 8
---- Counter::print() : end ----
&i: 0x22ccc8
sizeof(ptr): 4 sizeof(counter): 16
sizeof(int): 4 sizeof(double): 8
```

gcc 4.1.1 on fedora core 6

```
$ g++ -o app2 app2.cpp
$ ./app2
ptr = &counter: 0xbf841e20
counter.value(): 7
ptr->value(): 7
printCounter: counter value: 7
printByPtr: counter value: 7
---- Counter::print() : begin ----
my count_: 7
my address: 0xbf841e20
&count_ : 0xbf841e20 sizeof(count_): 4
&x_ : 0xbf841e24 sizeof(x_): 8
---- Counter::print() : end ----
&i: 0xbf841e1c
sizeof(ptr): 4 sizeof(counter): 12
sizeof(int): 4 sizeof(double): 8
```

- Different size of objects on different platform!
 - Different configuration of compiler
 - Optimization for access to memory
- Address of object is address of first data member in the object

Classes and Applications

- So far we have always included the definition of classes together with the main application in one file
- The advantage is that we have only one file to modify
- Disadvantage are many
 - There is always ONE file to modify no matter what kind of modification you want to make
 - This file becomes VERY long after a very short time
 - Hard to maintain everything in only one place
 - We compile everything even after very simple changes

Example of Typical Application So Far

```
// app3.cpp
#include <iostream>
using std::cout;
using std::endl;

#include "Counter.h"

Counter makeCounter() {
    Counter c;
    return c;
}

void printCounter(Counter& counter) {
    cout << "counter value: " << counter.value() << endl;
}

void printByPtr(Counter* counter) {
    cout << "counter value: " << counter->value() << endl;
}
```

```
int main() {
    Counter counter;
    counter.increment(7);

    Counter* ptr = &counter;

    cout << "counter.value(): " << counter.value()
         << endl;
    cout << "ptr = &counter: " << &counter << endl;
    cout << "ptr->value(): " << ptr->value() << endl;

    Counter c2 = makeCounter();
    c2.increment();

    printCounter( c2 );

    cout << "sizeof(ptr): " << sizeof(ptr)
         << " sizeof(c2): " << sizeof(c2)
         << endl;

    return 0;
}
```

Separating Classes and Applications

- It's good practice to separate classes from applications
- Create one file with only your application
 - Use `#include` directive to add all classes needed in your application
 - Keep a separate file for each class
- Compile your classes separately
- Include compiled classes (or libraries) when linking your application

First Attempt at Improving Code Management

```
// Datum1.cc
// include all header files needed
#include <iostream>
using namespace std;

class Datum {
public:
    Datum() { }

    Datum(double x, double y) {
        value_ = x;
        error_ = y;
    }

    Datum(const Datum& datum) {
        value_ = datum.value_;
        error_ = datum.error_;
    }

    void print() {
        cout << "datum: " << value_
              << " +/- " << error_
              << endl;
    }

private:
    double value_;
    double error_;
};
```

```
// app1.cpp
#include "Datum1.cc"

int main() {

    Datum d1;
    d1.print();

    Datum d2(0.23, 0.212);
    d2.print();

    Datum d3( d2 );
    d3.print();

    return 0;
}
```

```
$ g++ -o app1 app1.cpp
$ ./app1
datum: NaN +/- 8.48798e-314
datum: 0.23 +/- 0.212
datum: 0.23 +/- 0.212
```

Problems with Previous Example

- Although we have two files it is basically if we had just one!
- Datum1.cc includes not only the declaration but also the definition of class Datum
 - Implementation of all methods exposed to user
- When compiling app1.cpp we also compile class Datum every time!
 - We do not need any library because app1.cpp includes all source code!
 - When compiling and linking app1.cpp we also create compiled code for Datum to be used in our application
 - *Remember what #include does!*

Pre-Compiled version of Datum1.cc

- Our source file is only a few lines long

```
$ wc -l Datum1.cc
30 Datum1.cc
```

```
$ wc -l app1.cpp
16 app1.cpp
```

```
$ g++ -E -c Datum1.cc > Datum1.cc-precompiled
```

```
$ wc -l Datum1.cc-precompiled
23740 Datum1.cc-precompiled
```

- The precompiled version is almost 24000 lines!
 - This is all code included in and by iostream

```
$ grep "#include" /usr/lib/gcc/i686-pc-cygwin//3.4.4/include/c++/
iostream
```

```
* This is a Standard C++ Library header. You should @c #include
this header
```

```
#include <bits/c++config.h>
```

```
#include <ostream>
```

```
#include <istream>
```

iostream

```
#ifndef _GLIBCXX_IOSTREAM
#define _GLIBCXX_IOSTREAM 1

#pragma GCC system_header

#include <bits/c++config.h>
#include <ostream>
#include <istream>

namespace std
{
    /**
     * @name Standard Stream Objects
     */
    /*
    //@{
    extern istream cin;      ///< Linked to standard input
    extern ostream cout;    ///< Linked to standard output
    extern ostream cerr;    ///< Linked to standard error (unbuffered)
    extern ostream clog;    ///< Linked to standard error (buffered)

#ifdef _GLIBCXX_USE_WCHAR_T
    extern wistream wcin;   ///< Linked to standard input
    extern wostream wcout;  ///< Linked to standard output
    extern wostream wcerr;  ///< Linked to standard error (unbuffered)
    extern wostream wclog;  ///< Linked to standard error (buffered)
#endif
    //@}

    // For construction of filebuffers for cout, cin, cerr, clog et. al.
    static ios_base::Init __ioinit;
} // namespace std

#endif /* _GLIBCXX_IOSTREAM */
```

I have removed all comments from the file to make it fit in this slide

Additional code included by the header files in this file

How do you find **iostream** file on your computer?

Separating Interface from Implementation

- Clients of your classes only need to know the interface of your classes
- Remember:
 - Users should only rely on public members of your class
 - Internal data structure must be hidden and not needed in applications
- Compiler needs only the declaration of your classes, its functions and their signature to compile the application
 - Signature of a function is the exact set of arguments passed to a function and its return type
- The compiled class code (definition) is needed only at link time
 - Libraries are needed to link not to compile!

Header and Source Files

- We can separate the declaration of a class from its implementation
 - Declaration tells the compiler about data members and member functions of a class
 - We know how many and what type of arguments a function has by looking at the declaration but we don't know how the function is implemented
- Declaration of a class Counter goes into a file usually called Counter.h or Counter.hh suffix
- Implementation of methods goes into the source file usually called Counter.cc

Counter.h and Counter.cc

```
// Counter.h
// Counter Class: simple counter class.
// Allows simple or step
// increments and also a reset function

// include header files for types
// and classes used in the declaration

class Counter {
public:
    Counter();
    int value();
    void reset();
    void increment();
    void increment(int step);

private:
    int count_;
};
```

```
// Counter.cc
// include class header files
#include "Counter.h"

// include any additional header files
// needed in the class
// definition
#include <iostream>
using std::cout;
using std::endl;

Counter::Counter() {
    count_ = 0;
};

int Counter::value() {
    return count_;
}

void Counter::reset() {
    count_ = 0;
}

void Counter::increment() {
    count_++;
}

void Counter::increment(int step) {
    count_ = count_ + step;
}
```

Scope operator :: is used to tell methods belong to Class Counter

What is included in header files?

- Declaration of the class
 - Public and data members
- All header files for types and classes used in the header
 - data members, arguments or return types of member functions
- Sometimes when we have very simple methods these are directly implemented in the header file
- Methods implemented in the header file are referred to as inline functions
 - For example getter methods are a good candidate to become inline functions

What is included in source file?

- Header file of the class being implemented
 - Compiler needs the prototype (declaration) of the methods
- Implementation of methods declared in the header file
 - Scope operator `::` must be used to tell the compiler methods belong to a class
- Header files for all additional types used in the implementation but not needed in the header!
 - Nota bene: header files include in the header file of the class are automatically included in the source file

Compiling Source Files of a Class

```
$ g++ Counter.cc  
/usr/lib/gcc/i686-pc-cygwin/3.4.4/../../../../libcygwin.a(libcmain.o) : :  
undefined reference to `__WinMain@16'  
collect2: ld returned 1 exit status
```

WinXP+
cygwin

```
$ g++ Counter.cc  
/usr/lib/gcc/i386-redhat-linux/4.0.2/../../../../crt1.o(.text+0x18) :  
In function `__start': : undefined reference to `main'  
collect2: ld returned 1 exit status
```

Linux

- Do you understand the error?
- What does undefined symbol usually mean?
- Why we did not encounter this error earlier?