C++ Programming Basics

Andy R. Terrel

The Texas Advanced Computing Center

March 26, 2012

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

C++ is just an abomination. Everything is wrong with it in every way.

Jamie Zawinski
Netscape, 1994

C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it . . .

Linus Torvalds
Creator of Linux

. . . by and large I think it's a bad language. It does a lot of things half well and it's just a garbage heap of ideas that are mutually exclusive. Everybody I know, whether it's personal or corporate, selects a subset and these subsets are different.

Ken Thompson
Google

C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do it blows your whole leg off.

There are only two kinds of [programming] languages: the ones people complain about, and the ones nobody uses.

Bjarne Stroustrup
Creator of C++

C++ Standardization Timeline

- Began in 1979 at Bell Labs as "C with Classes"
- Renamed C++ in 1983
- Language standard ratified in 1998 as ISO/IEC 14882:1998
- Most recent amendment to the standard in 2003, ISO/IEC 14882:2003
- Next standard (formerly known as "C++0x") is in development, Final Committee Draft approved, possible standard finalization as early as 2011

C++ Technical Specifications

- C++ is a "multi-paradigm" language:
  - Imperative
  - Generic (templates)
  - Object Oriented
- This means that programmers can work in a variety of styles, freely intermixing constructs from different paradigms
- One can also write C++ code in the functional programming paradigm (think Haskell, Lisp) with template metaprogramming

- C++ is often said to be a "superset" of C, but

Superset of C $\not\Rightarrow$ Any C code will compile with a C++ compiler

- Simple example: C++ introduces new keywords, so legal C code such as:

```
int class = 0;
int public = 0;
int private = 0;
int new = 0;
```

will not compile with a C++ compiler.

- Knowledge of C is *absolutely not* required to begin programming in C++

- In fact, extensive C knowledge can be detrimental to learning C++ because:
  - You already "know" how to do things in C,
  - Therefore you are not exposed to the "C++ way" of doing them
  - Hence, you are still writing C code, but with a slower, more complicated compiler and worse binary compatibility!

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- There are only a few built-in data type categories in C/C++
  - `void`
  - `bool`
  - Integral types
  - Floating point types

- `void` is the anti-datatype
- You cannot declare a variable of type `void`

```
void x; // Error!
```

- Often used as a return "value" for functions which return nothing

- `bool` variables can only take on one of two values: `true` or `false`
- In C/C++, any zero- (resp. nonzero-) valued integral type is automatically convertible to false (resp. true)
- Could be implemented within a single *bit*, but its size is not specified by the standard and is usually 1 byte
- Exists primarily because, if it didn't, everyone would have an equivalent but differently-named boolean type

- The integral data types of C/C++ are
  - `char`
  - `int`
- Both have `unsigned` variations, signed is the default
- `int` has: `short`, `long`, and `long long` variations

- The standard guarantees *only* the size of `char`
- All other data type sizes are implementation-dependent (though most implementations are consistent)
- `sizeof(char)==1` (byte), guaranteed

- `char`:
  - Can represent integral values $\in [-128, 127]$[1]
  - Seven binary digits and one sign bit

- `unsigned char`:
  - Uses all 8 binary digits
  - Can represent values $\in [0, 255]$

[1] All ranges for integral types assume two's-complement arithmetic. This is very common, but not actually specified by the C++ standard.
`http://en.wikipedia.org/wiki/Two's_complement`

- One may drop the `int` keyword when referring to the different `int` variations, eg.

```
short i;
long j;
```

- The size of the different `int` variations is implementation defined, but *typically*

	`sizeof()`	max unsigned
`short`	2	65,535
`int`	4	4,294,967,295
`long`	4/8 (32/64-bit OS)	
`long long`	8	18,446,744,073,709,551,615

- The floating point data types of C/C++ are
  - `float` (aka single precision)
  - `double` (as in double precision)
- `double` has a "`long`" variation
- The standard guarantees only:

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

- Floating point numerical representations are quite complex!
- See: `http://en.wikipedia.org/wiki/Floating-point`

- When working with floating point numbers, we are primarily interested in a quantity called "machine epsilon"
- Machine epsilon is the difference between 1 and the least value greater than 1 that is representable
- C++ makes it easy to obtain the machine epsilon for any floating point data type with the `numeric_limits` class

`Type`	`sizeof()`	`numeric_limits<Type>::epsilon()`
`float`	4	1.19209e−07
`double`	8	2.22045e−16
`long double`	16	1.08420e−19

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

```
int y=10;
{
  // We can still see y
  int x=y;
}

// x is no longer accessible!
```

- Defined by any pair of "curly" brackets
- Used in functions, loops, or just normal code

```
int y=10;
{
  // We can still see y
  int x=y;
}

// x is no longer accessible!
```

- Perhaps the most ubiquitous "control structure"
- Because of this, also nearly invisible

```
void f(int x, double y)
{
  // function body, use x and y
}

f(1, 2.71828);
```

- Basic syntax is:

  `rtype fname(args) { stmt; return val; }`

  - `rtype` - the type of variable returned by the function
  - `fname` - name of the function
  - `args` - comma-separated list of arguments

```
void f(int x, double y)
{
  // function body, use x and y
}

f(1, 2.71828);
```

- Basic syntax is:

  `rtype fname(args) { stmt; return val; }`

  - `stmt` - code operating on `args`
  - `return` - C/C++ keyword: return to calling function
  - `val` - object of type `rtype`, returned to calling function

```
void f(int x, double y)
{
  // function body, use x and y
}

f(1, 2.71828);
```

- No distinction between "subroutine" and "function"
- Maximum of one return value allowed
- Functions must be declared (aka "prototyped") before they can be used

```
for (int i=0; i<N; ++i)
  {
    // Use i here
  }

// i is now "out of scope"
```

- Basic syntax is: `for (init; cond; inc)`
  - `init` - an expression initializing loop counters
  - `cond` - a boolean expression; stops loop execution when false
  - `inc` - an expression for updating the loop counter

```
for (int i=0; i<N; ++i)
  {
    // Use i here
  }

// i is now "out of scope"
```

- Same as the standard C for loop
- Loop variables may be initialized "in line"
- Here, `++i` and `i++` are equivalent to `i←i+1`. Prefer the former for non-trivial counters (`iterators`).

```
if (i > 0)
  { /* code for i > 0  */ }
else if (i < 0)
  { /* code for i < 0  */ }
else
  { /* code for i == 0 */ }
```

- Basic syntax is:

  `if (cond) {...} else {...}`

  - `cond` - a boolean expression; if true, code execution enters the first set of scoping braces, otherwise skips to the next set

```
int x=0;
switch (x)
  {
    case 0:
      // ...
      break;
    case 1:
      // ...
      break;
    default:
      // ...
  }
```

- Basic syntax is:

```
switch (var) {
  case val: { stmt; break; }
  default: {}
}
```

  - `var` - a variable of integral (`int`, `char`, `enum`, etc.) type.
  - `val` - A particular value which `var` can take. Must be of integral type.
  - `stmt` - Code which should be executed if `var==val`.

```
int x=0;
switch (x)
  {
    case 0:
      // ...
      break;
    case 1:
      // ...
      break;
    default:
      // ...
  }
```

- Basic syntax is:

```
switch (var) {
  case val: { stmt; break; }
  default: {}
}
```

  - `var` - a variable of integral (`int`, `char`, `enum`, etc.) type.
  - `val` - A particular value which `var` can take. Must be of integral type.
  - `stmt` - Code which should be executed if `var==val`.

- You must remember to `break` out of each case, otherwise *control will fall through to the next case*!

```
int x=0;
switch (x)
  {
    case 0:
      // ...
      break;
    case 1:
      // ...
      break;
    default:
      // ...
  }
```

- Basic syntax is:

```
switch (var) {
  case val: { stmt; break; }
  default: {}
}
```

  - `var` - a variable of integral (`int`, `char`, `enum`, etc.) type.
  - `val` - A particular value which `var` can take. Must be of integral type.
  - `stmt` - Code which should be executed if `var==val`.

- *Always* provide a `default` case to handle unexpected input

```
while (i > 0)
  { /* loop body */ }
```

- Basic syntax is:

  `while (cond) {...}`

  - `cond` - a boolean expression; loop execution continues while `cond` evaluates to true

- No formal mechanism to update the loop condition
- Something inside the body can make the condition false
- Can also exit with explicit `break`

```
do
  { /* loop body */ }
while (i > 0);
```

- Basic syntax is:

  `do {...} while (cond);`

  - `cond` - a boolean expression; loop execution continues while `cond` is true, loop body executes at least once

- Don't forget trailing semi-colon!

```
goto next_step;

// code that will
// be skipped...

next_step:

// Execution will
// begin again here
```

- Basic syntax is:

  ```
  goto label;
  ...
  label:
  ```

  - `label` - An arbitrary label marking a line in the source where code execution will jump when it hits the `goto` statement.

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- C++ I/O is handled via "stream" objects
- Standard C++ header files no longer have the .h extension

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- Namespaces prevent collisions between variables with the same name
- The `using` directive is a shortcut for prepending `std::`

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- The main program *must* be named "`main`" in both C and C++
- C99 and C++ standards require that `main` return `int`

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- cout is the "standard out" output stream in C++
- cout is a member of the `std::` namespace
- Alternatively, write "`std::cout << ...`" everywhere

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- The double less-than character "<<" is the stream "insertion operator"
- It performs a formatted output operation on the stream

C Hello World

```
#include <stdio.h>

int main()
{
  printf("Hello World\n");
  return 0;
}
```

C++ Hello World

```
#include <iostream>
using namespace std;
int main()
{
  cout << "Hello World\n";
  return 0;
}
```

- A zero return value typically indicates "success"
- Other programs and the OS itself may check the return status of your code

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- Both versions print: `3.1415926535897931e+00`
- *Many* programmers still prefer `printf` for its conciseness

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- C format expression is "%[width].[digits][type]"
- Possible values for "type" are: d, e, f, g, and others

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- `setprecision(N)`, `N` = number of digits after the decimal
- The `scientific` tag is equivalent to the "e" type in C

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- C format tags are specified in each `printf` call
- The C++ formatting tags affect subsequent writes as well

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- `endl` sends a newline character `'\n'` to the stream *and* flushes the stream's buffer

```
double pi = 3.14159265358979323846264338327950 29;
```

C Example

```
#include <stdio.h>
...
printf("%.16e\n", pi);
```

C++ Example

```
#include <iostream>
#include <iomanip>
using namespace std;
...
cout << setprecision(16)
     << scientific
     << pi << endl;
```

- `setprecision` is defined in the `<iomanip>` header

- One advantage of the C++ technique: Setting the format and creating the output can be effectively decoupled
- One can perform the same operations using "member function" syntax as well

```
#include <iostream>
using namespace std;
...
double pi = 3.14159265358979323846264338327950 29;

cout.precision(16);
cout.setf(ios::scientific);
cout << pi << endl;
```

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- Both versions print: 867-5309

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- In C, you must declare a separate buffer to hold the string
- There *must* be enough space in the buffer to hold *any* string which could be stored there

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- In C++, `ostringstream` is a stream whose output is a string
- It is defined in the standard `<sstream>` header

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- The `sprintf` function is a member of the `printf` family.
- Its first argument is the character buffer where the string is to be written

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- `ostringstream` uses stream insertion just like other streams
- Formatting of the types can be automatic or controlled with the stream formatting tags discussed previously

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- The `str()` member function returns a copy of the output `string` that was created by stream insertion

C Example

```
#include <stdio.h>
...
char buffer[16];
int i=867, j=5309;
sprintf(buffer,
  "%d-%d", i, j);
printf("%s\n", buffer);
```

C++ Example

```
#include <iostream>
#include <sstream>
using namespace std;
...
ostringstream s;
int i=867, j=5309;
s << i << '-' << j;
cout << s.str() << endl;
```

- The primary benefit of the C++ version is that memory is managed automatically

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

- C++ file "streams" are defined in the "<fstream>" header

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

- In C, a `FILE*` pointer is used to interface with files
- We must also specify the access mode "w", for writing

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

- In C++, resource acquisition is initialization: construction of the `ofstream` object coincides with opening the file
- The type of access is automatically determined

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

- In C, a different function with a different signature is required to write to file (`fprintf` vs. `printf`)
- In C++, the same stream insertion operator is reused

Write to file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","w");
fprintf(fptr,
  "Hello World\n");
fclose(fptr);
```

Write to file in C++

```
#include <fstream>
using namespace std;
...
ofstream file("file.txt");
file << "Hello World\n";
```

- In C, you *must* remember to close the file, otherwise the resource will be "leaked"
- In C++, the file is closed automatically when the object goes out of scope

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- Assuming we now read the same file back in . . .

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- In C, mode `"r"` is used to specify file read access
- The `ifstream` object opens files for reading automatically

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- `fgets` reads a single line (up to 32 characters, here) from file
- The newline character (`'\n'`) is retained in `buf`

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- In C++, the `string` object is used to interact with character strings

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- The `getline` function reads an entire line from a stream
- Memory for storing for the string is handled internally

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- The newline character is extracted from the stream and *discarded*

Read from file in C

```
#include <stdio.h>
...
FILE *fptr =
  fopen("file.txt","r");
char buf[32];
fgets(buf, 32, fptr);
printf("Read: %s", buf);
fclose(fptr);
```

Read from file in C++

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
...
ifstream file("file.txt");
string s;
getline(file, s);
cout << "Read: "
     << s << endl;
```

- Both codes print "Read: Hello World"

C Example

```
#include <stdio.h>
...
int n;
printf("Enter any integer:\n");
scanf("%d", &n);
...
```

C++ Example

```
#include <iostream>
using namespace std;
...
int n;
cout << "Enter any
  integer:\n";
cin >> n;
...
```

- Processing user input from stdin is not common, but it's worth knowing how to do

C Example

```
#include <stdio.h>
...
int n;
printf("Enter any integer:\n");
scanf("%d", &n);
...
```

C++ Example

```
#include <iostream>
using namespace std;
...
int n;
cout << "Enter any
  integer:\n";
cin >> n;
...
```

- The `scanf` function will pause to wait for user input
- In C++, the "extraction operator" >> is used to get information out of the standard input stream, `cin`.

1. Write a simple program, e.g. "Hello World", compile it, run it, and verify the output on your system.

2. Enhance your simple program by adding a function, e.g. `hello_world()`, defined before `main()`. Call this function from `main()`. *Bonus: create your function in a separate text file from* `main()`*, and learn how to compile and link programs with multiple object files.*

3. Add a second function, say `goodbye_world()`, to your sample program. Read input from the user, and depending on that input, call one of the functions you've defined.

4. Write a code which prints out a "table" of floating point values. Format the output so there are 5 space-separated numbers per line, and the numbers are in "fixed" format with a width of 8 digits.

Hint: you may want to generate random data to print, see 'man random' for details of the pseudo-random number generator available in C/C++.

```
double rv =
  static_cast<double>(random()) /
  static_cast<double>(RAND_MAX);
```

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- Pointers have been around since the beginning of C:
  - Refer to another value stored in memory using its address
  - To retrieve the referred-to value is to "dereference the pointer"
  - May take special value "`NULL`" meaning "points to nothing"

```
int  i = 5;
int *p = &i;
int *n = NULL;
cout << i  << endl;
cout << p  << endl;
cout << *p << endl;
cout << n  << endl;
```

Prints:

```
5
0xbfffee18
5
0
```

- Pointers are absolutely essential in C and C++ because both languages use "pass by value" semantics

What do you think will be printed?

```
void increment(int x)
{ x = x + 1; }

int main()
{
  int i = 99;
  increment(i);
  cout << i << endl;
  return 0;
}
```

- Pointers are absolutely essential in C and C++ because both languages use "pass by value" semantics

```
void increment(int x)
{ x = x + 1; }

int main()
{
  int i = 99;
  increment(i);
  cout << i << endl;
  return 0;
}
```

What do you think will be printed?

If you guessed 99, you are correct!

- Pointers are absolutely essential in C and C++ because both languages use "pass by value" semantics

```
void increment(int x)
{ x = x + 1; }

int main()
{
  int i = 99;
  increment(i);
  cout << i << endl;
  return 0;
}
```

What do you think will be printed?

If you guessed 99, you are correct!

A *copy* of `i` is passed to the `increment` function, any changes made within the function body are *not* propagated to the calling function.

- Passing a pointer, on the other hand, allows the original to be modified: the copy still refers to the original location in memory.

```
void increment(int *x)
{ *x = *x + 1; }

int main()
{
  int i = 99;
  increment(&i);
  cout << i << endl;
  return 0;
}
```

This time, it will print 100.

- Even more important: "pass by value" implies a copy, and this copy could be *very* expensive depending on the data type!

Right Way

```
void f(BigData* x)
{ ... }

int main()
{
  BigData big;
  f(&big);
  return 0;
}
```

Wrong Way!

```
void f(BigData x)
{ ... }

int main()
{
  BigData big;
  f(big);
  return 0;
}
```

In addition to (and to complement) pointers, C++ provides *references*:

- A reference is an *alternative name* for an object
- References *cannot* be `NULL` – they always refer to something
- References *cannot* be "reseated," that is, assigned to refer to a different object, after they have been created

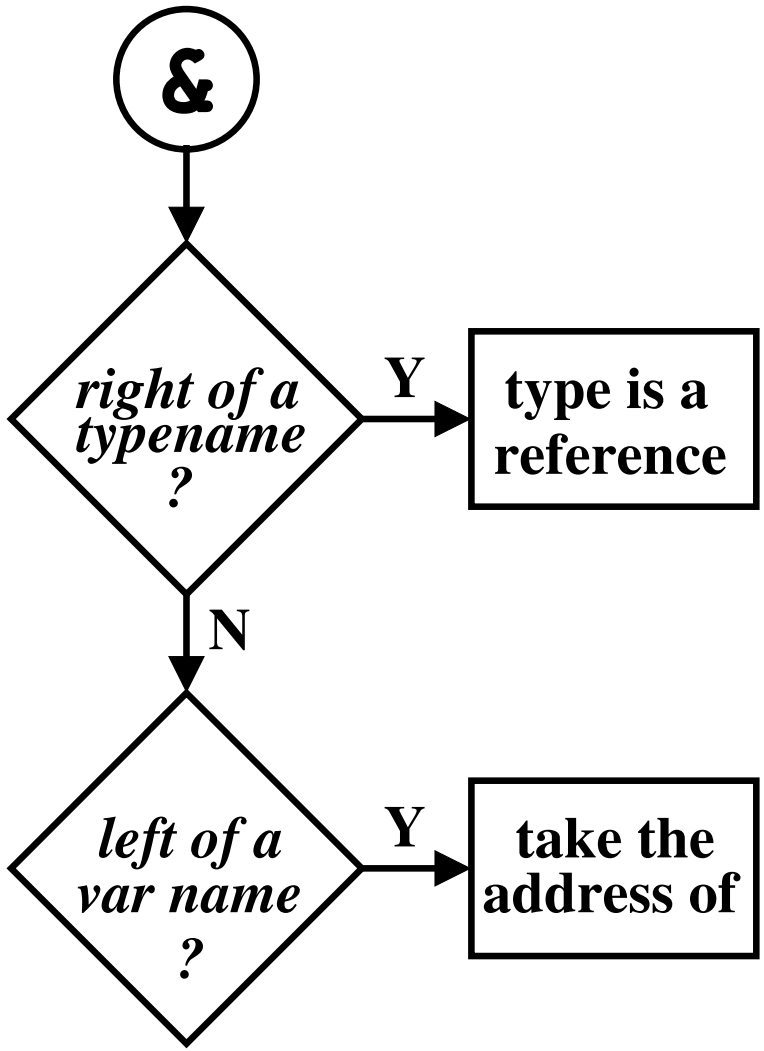
- C pointer syntax was "recycled" to implement references
- This causes a great deal of confusion to new users

```
int  i = 5;
int &r = i;
cout << i  << endl;
cout << r  << endl;
cout << &i << endl;
cout << &r << endl;
```

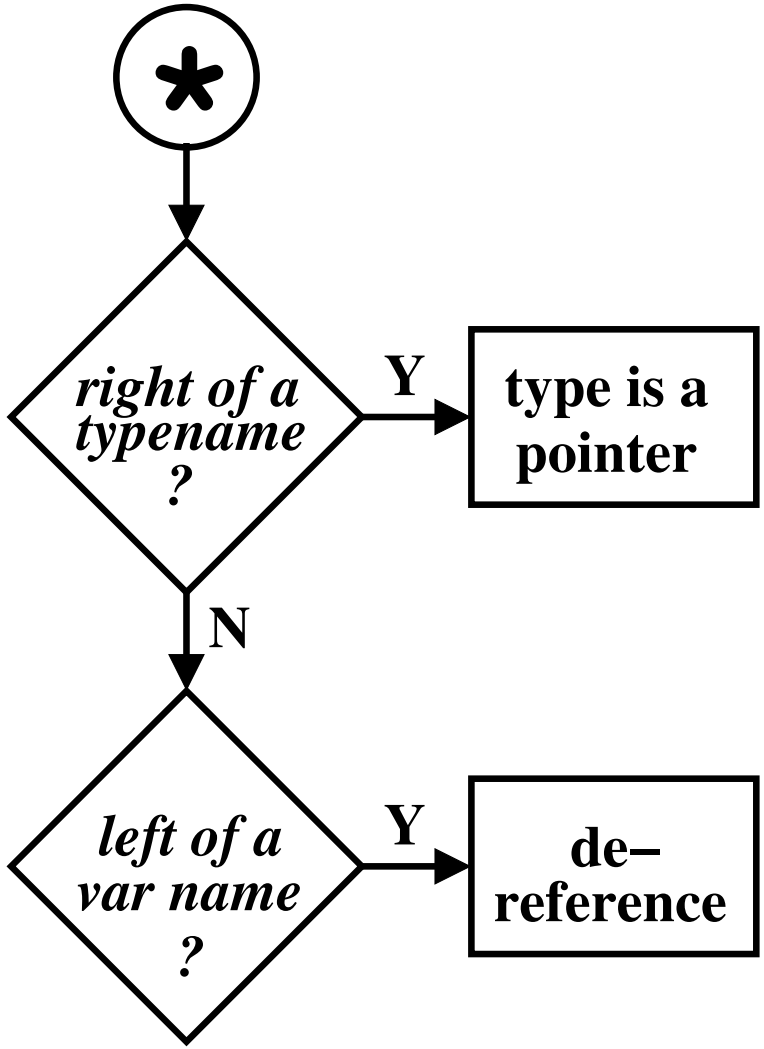
Prints:

```
5
5
0xbfffee0c
0xbfffee0c
```

- What is the deal with all these & and ∗ characters?
- It depends on the context!
- Use these simple flowcharts to help you remember



- What is the deal with all these & and ∗ characters?
- It depends on the context!
- Use these simple flowcharts to help you remember



Some Examples:

```
int *p = &i;
```

"p is of type pointer-to-int, it is equal to the address of i"

```
int &r = i;
```

"r is of type reference-to-int, it refers to the integer i"

```
cout << *i << endl;
```

"dereference i, print the value it refers to"

```
int main(int argc,
         char** argv)
```

"main is a function taking an int called argc, and a pointer to a pointer to a char, called argv"

- With references, we have one more acceptable way to pass arguments to functions

Pass by Pointer

```
void f(BigData* x)
{ x->value = ... }

int main()
{
  BigData big;
  f(&big);
  return 0;
}
```

Pass by Reference

```
void f(BigData& x)
{ x.value = ... }

int main()
{
  BigData big;
  f(big);
  return 0;
}
```

- Consider the following code:

```
#include <iostream>
using namespace std;
void f(int* p)
{ cout << *p << endl; }

int main()
{
  int array[4] =
   {42, -1, 9, 0};
  f(array);
  return 0;
}
```

1. Will this code even compile?
2. What is passed to the function `f`?
3. What will the function `f` print?

- Consider the following code:

```
#include <iostream>
using namespace std;
void f(int* p)
{ cout << *p << endl; }

int main()
{
  int array[4] =
   {42, -1, 9, 0};
  f(array);
  return 0;
}
```

1. Will this code even compile?
2. What is passed to the function `f`?
3. What will the function `f` print?

---

1. Yes!

- Consider the following code:

```
#include <iostream>
using namespace std;
void f(int* p)
{ cout << *p << endl; }

int main()
{
  int array[4] =
   {42, -1, 9, 0};
  f(array);
  return 0;
}
```

1. Will this code even compile?
2. What is passed to the function `f`?
3. What will the function `f` print?

---

1. Yes!
2. A pointer to the *beginning* of the array

- Consider the following code:

```
#include <iostream>
using namespace std;
void f(int* p)
{ cout << *p << endl; }

int main()
{
  int array[4] =
   {42, -1, 9, 0};
  f(array);
  return 0;
}
```

1. Will this code even compile?
2. What is passed to the function `f`?
3. What will the function `f` print?

---

1. Yes!
2. A pointer to the *beginning* of the array
3. 42, the value at the beginning of the array

- The code in the previous example works because arrays *automatically degrade to pointers* when passed to functions expecting pointers.
- Therefore, you need not worry about accidentally passing an array by value, and you don't have to take the address.
- Aside: Arrays can have any name, they need not be called `array`, as in this example!

- So, arrays and pointers are the same thing then?
- Not exactly, consider:

```
int a[4] =
   {42, -1, 9, 0};
int *p;

p = a; // OK
a = p; // Error!
```

- A pointer, like "p" above, is a variable, and can be assigned to point anywhere.
- An array name, like "a" above, is *not* a variable, thus the converse is an error.

- Since a pointer can point anywhere, it can also point somewhere in the middle of an array
- Moving a pointer is accomplished with simple addition and subtraction, aka "pointer arithmetic"

```
int a[4] =
   {42, -1, 9, 0};

int *p = a;
cout << *p << endl;
p+=3; // Equivalent to p=p+3;
cout << *p << endl;
```

Prints:

42

0

- Array access is handled with the "square brackets" notation

```
int a[4] =
   {42, -1, 9, 0};

cout << a[2] << endl;
```

Prints 9

- This fact, plus the fact that addition is a commutative operation, leads to a notational curiosity:

```
a[2] == *(a+2) == *(2+a) == 2[a]
```

```
int a[4] =
   {42, -1, 9, 0};

cout << 2[a] << endl;
```

Prints 9. Really!

- Never write code this way
- Consider it a useful mnemonic device for remembering: "array access is pointer arithmetic"

- Hopefully by now you are convinced that arrays and pointers are not really the same thing, except that they are, sometimes.
- Now you are almost as confused as all other C/C++ programmers!

- Our simple examples have dealt with fixed-size arrays thus far
- In general, you want to decide *at runtime* how large an array should be
- The C89 standard did not allow for variable length arrays (VLAs)
- C99 does, and all modern C compilers now support VLAs

- Some drawbacks of arrays:
  - Arrays do not generally "know" their own size[2]
  - Arrays, even the VLAs of C99, cannot be resized
  - C99 does not specify whether VLAs shall be stack or heap variables
- In a C talk, we would start discussing dynamic memory allocation, then memory leaks, then debugging ...
- Luckily for us, we can move straight to `std::vector` instead!

[2] If you know the type stored in the array (eg. `int`), you can always use the `sizeof(array)/sizeof(int)` calculation to determine the array's size

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- Vector is declared in the standard header, <vector>

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- <int> is a template argument
- Defines the type to be stored in the vector

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- v is the name of the vector
- v is constructed with size 2
- We say: "v is a vector of int"

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- A vector always knows its own size
- Accessible through the `size()` member function

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- The `resize()` member function changes the vector's size
- `vector` is guaranteed to be contiguous in memory, just like an array

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- Vector access uses square brackets, just like array
- Unlike array, square brackets *are not* shorthand for pointer arithmetic

```
#include <iostream>
#include <vector>
using namespace std;
...
vector<int> v(2);
cout << v.size() << endl;
v.resize(4);
cout << v.size() << endl;
cout << v[1] << endl;
```

Prints:

```
2
4
0
```

- Entries of the vector are initialized (default constructed)
- All built-in types (`int`, `double`, etc.) have default value 0

```
#include <vector>
#include <iostream>
using namespace std;
...
vector<int> v;
v.reserve(100);
cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100
```

- Efficient use of `vector` requires minimizing the number of calls to `resize()`

```
#include <vector>
#include <iostream>
using namespace std;
...
vector<int> v;
v.reserve(100);
cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100
```

- We don't have to specify a size at construction time

```
#include <vector>
#include <iostream>
using namespace std;
...
vector<int> v;
v.reserve(100);
cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100
```

- The `reserve()` member pre-allocates memory

```
#include <vector>
#include <iostream>
using namespace std;
...
vector<int> v;
v.reserve(100);
cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100
```

- This does not change the *size* of the `vector`

```
#include <vector>
#include <iostream>
using namespace std;
...
vector<int> v;
v.reserve(100);
cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100
```

- The capacity() member function returns the current maximum capacity of the vector

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

v.push_back(1123);

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

0
100

1
100

- Continuing from the previous slide . . .

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

v.push_back(1123);

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100


1
100
```

- The `push_back()` member "pushes" entries onto the "back" of the vector

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

v.push_back(1123);

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100


1
100
```

- For each entry added with `push_back()`, the `size` of the `vector` is increased by 1

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

v.push_back(1123);

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
0
100

1
100
```

- If the `vector` has enough capacity, `push_back()` uses existing space, otherwise it increases the capacity

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

cout << v[10] << endl;
v.clear();

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
1
100

?


0
100
```

- Continuing from the previous slide . . .

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

cout << v[10] << endl;
v.clear();

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
1
100

?


0
100
```

- Attempting to access an entry past the "end" causes undefined behavior (UB), *even if there is enough capacity*
- UB: anything can happen, the program may even crash
- *Don't do this!*

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

cout << v[10] << endl;
v.clear();

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
1
100

?


0
100
```

- The `clear()` member "erases" all the elements of a vector
- If the elements of the `vector` are *themselves* pointers, the pointed-to values are *not touched in any way*.
- Be careful! This is a very easy way to leak memory!

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

cout << v[10] << endl;
v.clear();

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
1
100

?


0
100
```

- `clear()` also resets the `size` of the `vector` to 0 ...

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

cout << v[10] << endl;
v.clear();

cout << v.size() << endl;
cout << v.capacity() << endl;
```

Prints:

```
1
100

?


0
100
```

- . . . but it *does not* change the `vector`'s capacity!

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

vector<int>().swap(v);

cout << v.capacity() << endl;
```

Prints:

0
100

0

- Reducing `vector` capacity is not straightforward
- C++ designers wanted to make it hard to accidentally introduce poor performance
- Note: memory is automatically reclaimed when a `vector` goes out of scope

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

vector<int>().swap(v);

cout << v.capacity() << endl;
```

Prints:

0
100

0

- The trick is to swap your vector with an empty one
- The vector<int>() syntax creates an unnamed temporary variable
- The vector's swap() member function is subsequently called

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

vector<int>().swap(v);

cout << v.capacity() << endl;
```

Prints:

0
100

0

- The unnamed temporary immediately goes "out of scope"
- `vector::swap()` is *very* efficient: only a few pointers have to be swapped

```
...
cout << v.size() << endl;
cout << v.capacity() << endl;

vector<int>().swap(v);

cout << v.capacity() << endl;
```

Prints:

0
100

0

- Finally, we are left with the capacity of the default constructed zero-length vector
- The funny syntax is an indication you shouldn't have to do this very often

C API

```
void f(int *x, int n)
{
  for (int i=0; i<n; ++i)
    // modify x[i] ...
}
...
```

C++ code, calling C API

```
#include <vector>
using namespace std;
int main()
{
  vector<int> v(10);
  f(&v[0], v.size());
  ...
  return 0;
}
```

- vector is designed to work seamlessly with C APIs expecting raw pointers

C API

```
void f(int *x, int n)
{
  for (int i=0; i<n; ++i)
    // modify x[i] ...
}
...
```

C++ code, calling C API

```
#include <vector>
using namespace std;
int main()
{
  vector<int> v(10);
  f(&v[0], v.size());
  ...
  return 0;
}
```

- To do so, we pass *the address* of the first element of the `vector`

C API

```
void f(int *x, int n)
{
  for (int i=0; i<n; ++i)
    // modify x[i] ...
}
...
```

C++ code, calling C API

```
#include <vector>
using namespace std;
int main()
{
  vector<int> v(10);
  f(&v[0], v.size());
  ...
  return 0;
}
```

- Be very careful! If the C API performs dynamic memory allocation at the pointer, the results are undefined!

- If you know the desired size, `N`, at construction time, create your `vector` with

```
vector<type> v(N);
```

- If you know the desired size, `N`, at construction time, create your `vector` with

```
vector<type> v(N);
```

- If you know exactly how much space is needed at any time after construction...

```
v.resize(N);
```

- If you know the desired size, `N`, at construction time, create your `vector` with

```
vector<type> v(N);
```

- If you know exactly how much space is needed at any time after construction...

```
v.resize(N);
```

- If you can estimate how much space will be required, reserve that space and `push_back()` entries

```
v.reserve(N);
v.push_back(...);
```

- If you know the desired size, `N`, at construction time, create your `vector` with

```
vector<type> v(N);
```

- If you know exactly how much space is needed at any time after construction...

```
v.resize(N);
```

- If you can estimate how much space will be required, reserve that space and `push_back()` entries

```
v.reserve(N);
v.push_back(...);
```

- If you have no idea how much space you'll need, all you can do is `push_back()`.

- We covered:
  - `vector` construction, `size()`, `operator[]`, `resize()`, `reserve()`, `capacity()`, `push_back()`, `clear()`, and `swap()`
  - Using `vector` with "C" APIs
- In general: *always* prefer `std::vector` over arrays!
- Not covered in detail: the Standard Template Library
  - `vector` is only a single component of the Standard Template Library (STL)
  - Familiarity with the STL is essential to maximizing your productivity in C++

1. As hinted at previously, the `main()` function can take two standard arguments

```
int main(int argc, char** argv)
{
  ...
  return 0;
}
```

1. As hinted at previously, the `main()` function can take two standard arguments

```
int main(int argc, char** argv)
{
  ...
  return 0;
}
```

   - `argc` is the number of command line arguments, including the program name
   - `argv` is a pointer to an array of "strings" in the form of `char*`s

1. As hinted at previously, the `main()` function can take two standard arguments

```
int main(int argc, char** argv)
{
  ...
  return 0;
}
```

Write code to print out the command line args, and test your code by running it with different numbers and types of command line arguments. *Hint: you can access the individual arguments via* `argv[0]`, `argv[1]`, ...

2. Write a function which resizes and fills a passed `vector` with values. Back in the `main()` program, print out the values in the vector to ensure they are the same values set by the function.

3. Create a "matrix"-like object using a

```
vector<vector<double> > matrix;
```

3. Create a "matrix"-like object using a

```
vector<vector<double> > matrix;
```

Note that the space between the trailing double greater-than signs is *required* to allow the compiler to parse this statement correctly.

3. Create a "matrix"-like object using a

```
vector<vector<double> > matrix;
```

Write code to:

a.) Resize your matrix to hold 100×100 elements
b.) Fill your matrix with values
c.) Multiply your matrix by a vector or another matrix
d.) Compare the "performance" of your matrix operations with similar operations on a

```
double array[100][100] da;
```

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- The first objects in C were called `structs`
- Most common use: *to combine multiple basic data types into a single type*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
};
```

- The first objects in C were called `structs`
- Most common use: *to combine multiple basic data types into a single type*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
};
```

- Don't forget semi-colon at end of `struct` definition!

- The first objects in C were called `structs`
- Most common use: *to combine multiple basic data types into a single type*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
};
```

- It is generally considered bad practice to put "`using namespace std;`" directives in headers

- We can now work with "objects of type `Grid`" instead of two separate `vectors`, `x` and `y`

main.C

```
#include "grid.h"
int main()
{
  std::vector<Grid> vg;
  ...
  return 0;
}
```

- Consider how one might use the `Grid` object:

```
#include "grid.h"
...
Grid g;
g.x.resize(10);
g.y.resize(10);
...
```

- This works, but
  - The user must know the type and purpose of `x` and `y`
  - We rely on the user to keep the sizes of `x` and `y` synchronized
  - Code changes over time (what if a `z vector` were added to `Grid`?)

- An object-oriented (OO) approach may address these difficulties by adding *member functions*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- An object-oriented (OO) approach may address these difficulties by adding *member functions*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- The "unsigned" type is equivalent to "unsigned int"

- An object-oriented (OO) approach may address these difficulties by adding *member functions*

grid.h

```
#include <vector>
struct Grid
{
  std::vector<double> x, y;
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- Alerts the user that the argument should not be negative

- `Grid` usage might now look like this:

```
#include "grid.h"
...
Grid g;
g.resize(10);
...
```

- An improvement, however:
  - A "clever" or misinformed user can still modify `x` and `y` directly
  - Documentation may help some, but compiler-enforced access control is better

grid.h

```
#include <vector>
struct Grid
{
private:
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- C++ introduces the `private` and `public` keywords to enforce access control

grid.h

```
#include <vector>
struct Grid
{
private:
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- Declaring `x` and `y` "`private`" prevents clients of the object from using them directly

grid.h

```
#include <vector>
struct Grid
{
private:
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- The `resize()` interface is declared "`public`"
- Indicates how you intend the object to be used

grid.h

```
#include <vector>
struct Grid
{
private:
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- You can have as many `public` and `private` sections as desired

```
#include "grid.h"
...
Grid g;
g.x.resize(10); // Error!
...
```

- This type of usage is now prevented by the compiler
- The error message will be something along the lines: "x is private within this context"

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- Almost all C++ programmers actually use the `class` keyword instead of `struct`

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- The main difference is that `struct` has default `public` access while `class` has default `private`

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- Some developers reserve the use of `struct` to mean a C-style struct having no member functions

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  void resize(unsigned n)
    { x.resize(n); y.resize(n); }
};
```

- At the very least, use of the `class` keyword is a good way to indicate your *intent* to use C++-specific features in the object

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- C++ provides specially named member functions called "constructors" for creating user-defined objects
- Constructors do several useful things ...

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

1. Provide a *well-defined* procedure for creating an object
2. Enforce "resource acquisition is initialization" (RAII) paradigm
3. Allow efficient object creation
4. Require less typing

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- Constructors *always* have the same name as the object itself

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- Constructors can take an argument list

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- If you provide default values for all constructor arguments, your class can be "default-constructed"
- That is, code like

  ```
  Grid g;
  vector<Grid> vg(10);
  ```

  will work.

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- The initialization list begins with a colon immediately after the argument list
- It is a comma-separated list which calls the constructors for data members of the class
- You must initialize data members *in the same order* as they are defined in the class

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- The constructor function body may be empty (as here) or perform further initialization

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- A `class` may have any number of constructors
- When you don't provide *any* constructors, the compiler supplies the default constructor for you automatically

grid.h

```
#include <vector>
class Grid
{
  std::vector<double> x, y;
public:
  Grid(unsigned n=0)
    : x(n), y(n) {}
...
};
```

- Classes with dynamically allocated memory should also provide a destructor.
- The destructor also has the same name as the class, preceded by a tilde "~" character, eg. `~Grid()`.

- One of the biggest benefits of the OO programming paradigm is *systematic code reuse* through inheritance
- Objects related via inheritance are variously called
  - "Base" / "Derived"
  - "Parent" / "Child"
  - "Superclass" / "Subclass"

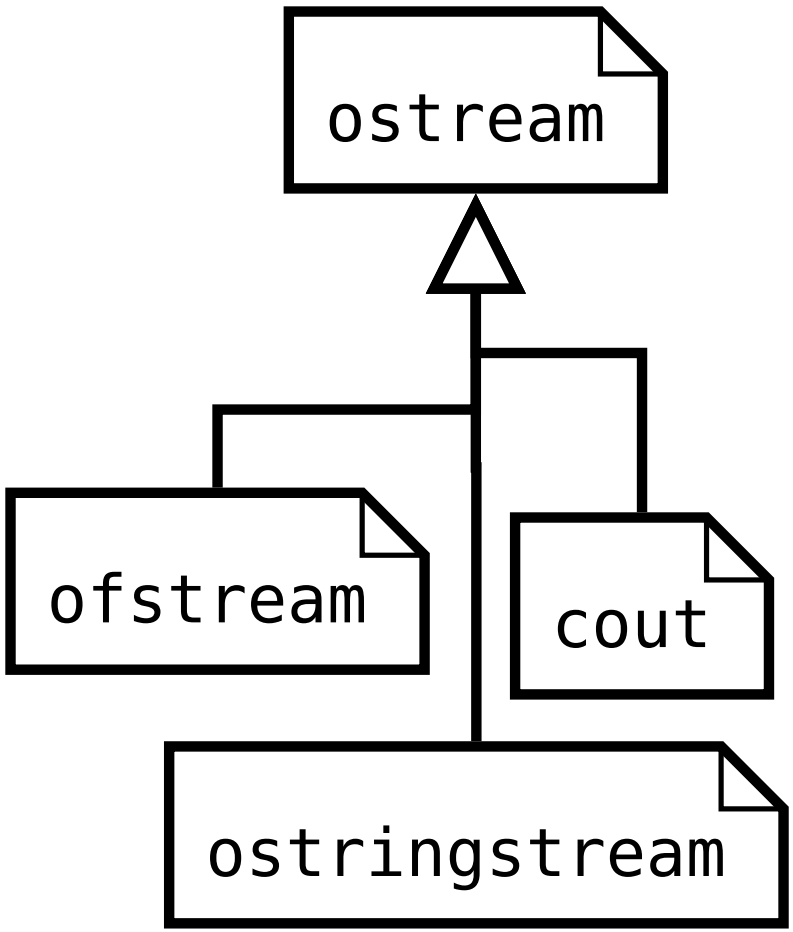
  pairs
- "Polymorphism" is the property which allows objects of type `Derived*` to be referred to via objects of type `Base*`[3]

[3] Likewise for `Derived&`/`Base&` pairs.

- An example will help tie together inheritance with everything else we have learned thus far

- Consider: the standard `ofstream`, `ostringstream`, and `cout` objects are all derived from a common base class, known as `std::ostream`

- Inheritance is sometimes expressed with diagrams
- Unified Modeling Language (UML)
- Arrows point *from* child *to* parent



print.h

```
#include <iostream>

void print(std::ostream& out)
{
  out << "Hello World\n";
}
```

- Inheritance allows us to write functions whose arguments are pointers or references to *Base* classes
- The print function does not know whether the *actual* stream object is a file, string, or even stdout

```
#include "print.h"

int main()
{
  // Print to stdout
  print(std::cout);
}
```

- By passing cout to the print function, we can print to stdout

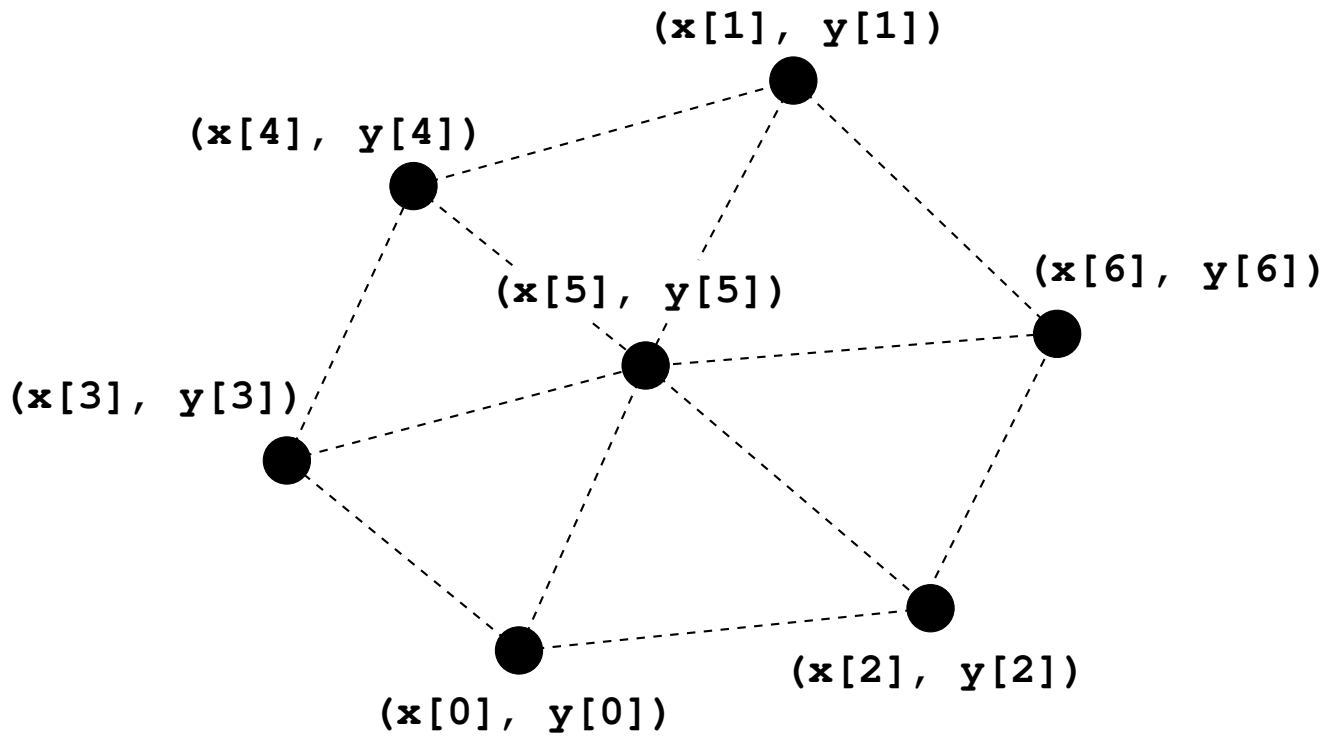
```
#include "print.h"
#include <fstream>
using namespace std;

int main()
{
  // Print to file
  ofstream file("file.txt");
  print(file);
}
```

- The same function can be passed an `ofstream` object, and it will print to file

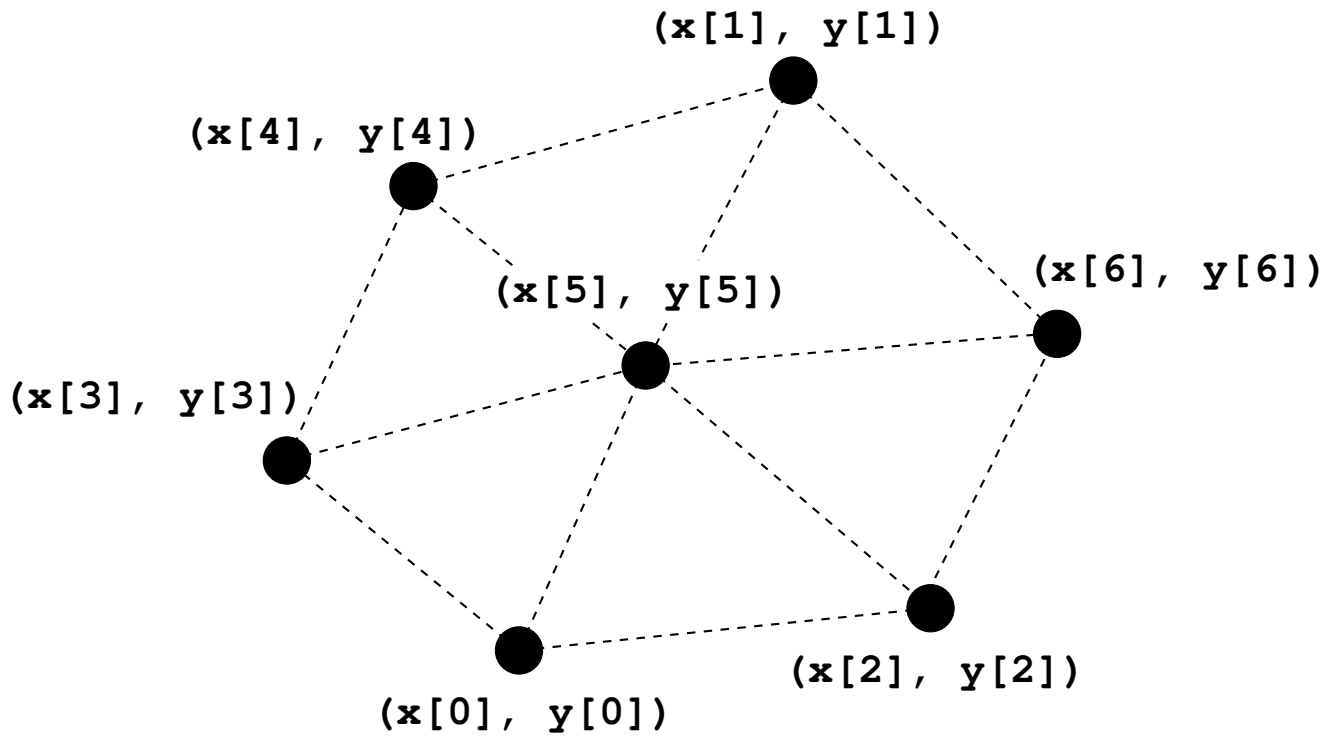
```
#include "print.h"
#include <sstream>
using namespace std;

int main()
{
  // Perform formatted
  // write to string
  ostringstream s;
  print(s);
}
```

- And finally, the same function can also be used to print to a string stream
- This example is trivial
- Code reuse, enabled through polymorphism, is essential to maintaining an organized software project!

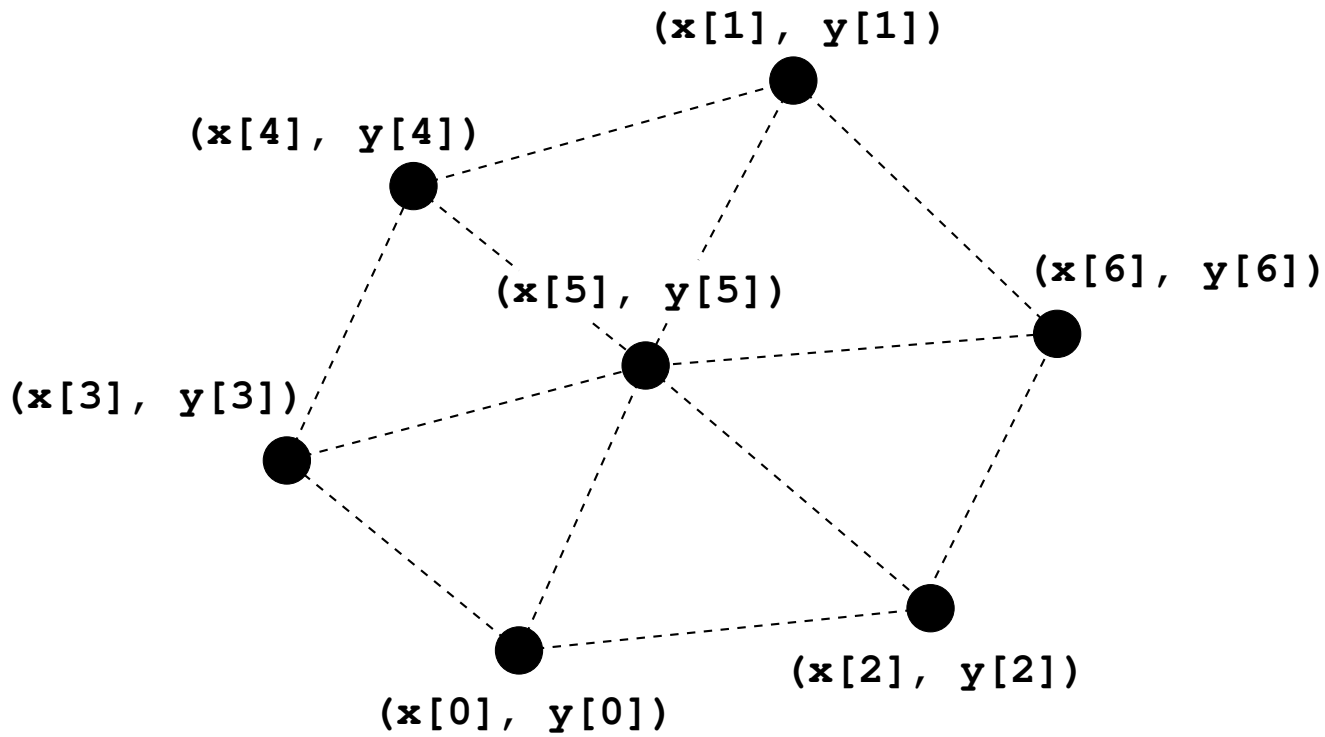
- Consider again the `Grid` class
- Our initial design had an "unstructured" grid of points in mind
- How can we implement inheritance in this example?



- Assume there are `n` total points



- For a given `m < n`, the coordinates of point `m` are given by: `(x[m], y[m])`



- The dashed lines *suggest* a possible connectivity, but we don't discuss connectivity in this example

grid.h

```
#include <vector>

class Grid
{
  std::vector<double> _x, _y;
public:
  double& x(unsigned m)  { return _x[m]; }
  double& y(unsigned m)  { return _y[m]; }
  unsigned n() { return _x.size(); }
};
```

- Let's start by making a few changes to the `Grid` class
- This will make inheriting from `Grid` easier

grid.h

```
#include <vector>

class Grid
{
  std::vector<double> _x, _y;
public:
  double& x(unsigned m)  { return _x[m]; }
  double& y(unsigned m)  { return _y[m]; }
  unsigned n() { return _x.size(); }
};
```

- It's common to name `private` members differently so they are easier to identify

grid.h

```
#include <vector>

class Grid
{
  std::vector<double> _x, _y;
public:
  double& x(unsigned m)  { return _x[m]; }
  double& y(unsigned m)  { return _y[m]; }
  unsigned n() { return _x.size(); }
};
```

- We've also added "accessor" functions
- Use to set the values in the _x and _y vectors

grid.h

```
#include <vector>

class Grid
{
  std::vector<double> _x, _y;
public:
  double& x(unsigned m)  { return _x[m]; }
  double& y(unsigned m)  { return _y[m]; }
  unsigned n() { return _x.size(); }
};
```

- One should also check the input value `m`!
- All input checking is omitted here, to save space

grid.h

```
#include <vector>

class Grid
{
  std::vector<double> _x, _y;
public:
  double& x(unsigned m)  { return _x[m]; }
  double& y(unsigned m)  { return _y[m]; }
  unsigned n() { return _x.size(); }
};
```

- We've also added a function which can tell us the current size of the data vectors

grid.h

```
#include <vector>
#include <utility>
class Grid
{
  std::vector<double> _x, _y;
public:
  typedef std::pair<double, double> Point;
  Point get_point(unsigned m)
    { return Point(_x[m], _y[m]); }
};
```

- We also define the `Point` type to simplify interactions with the class

grid.h

```
#include <vector>
#include <utility>
class Grid
{
  std::vector<double> _x, _y;
public:
  typedef std::pair<double, double> Point;
  Point get_point(unsigned m)
    { return Point(_x[m], _y[m]); }
};
```

- The syntax for defining a type is:
  `typedef existing_type new_type;`

grid.h

```
#include <vector>
#include <utility>
class Grid
{
  std::vector<double> _x, _y;
public:
  typedef std::pair<double, double> Point;
  Point get_point(unsigned m)
    { return Point(_x[m], _y[m]); }
};
```

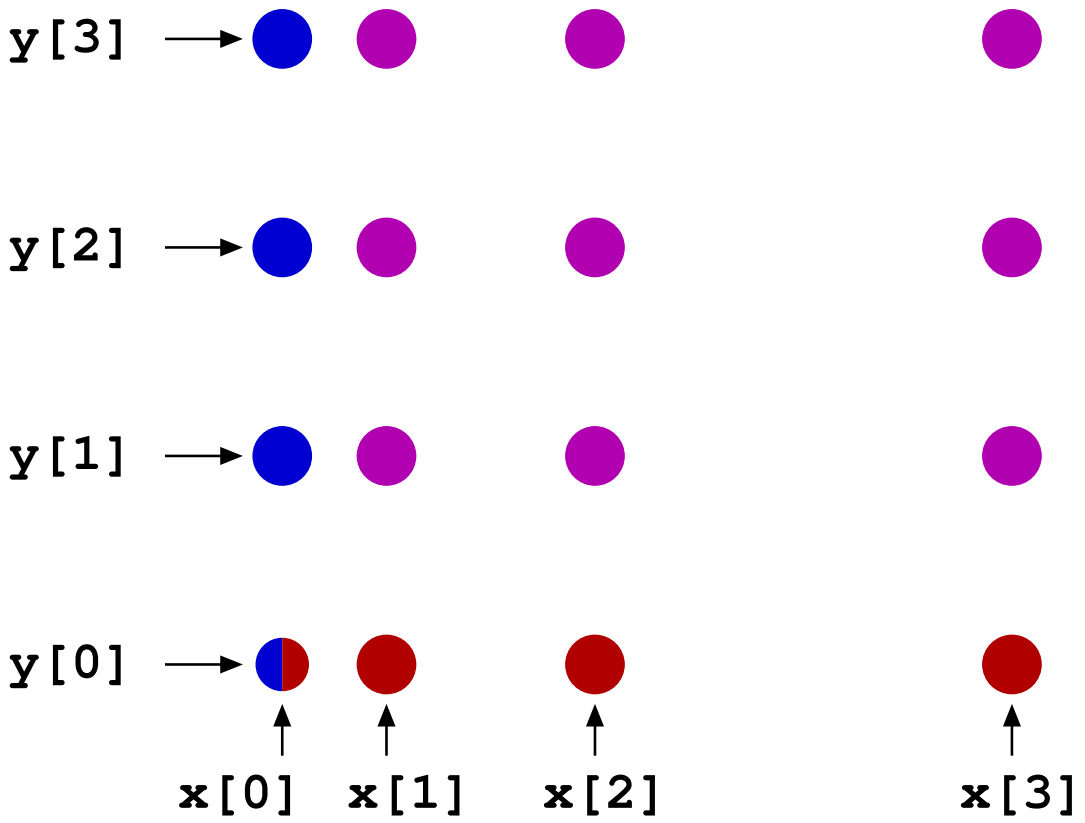
- The `pair` type is defined in the standard `<utility>` header
- `pair` is templated on two (possibly heterogeneous) types

grid.h

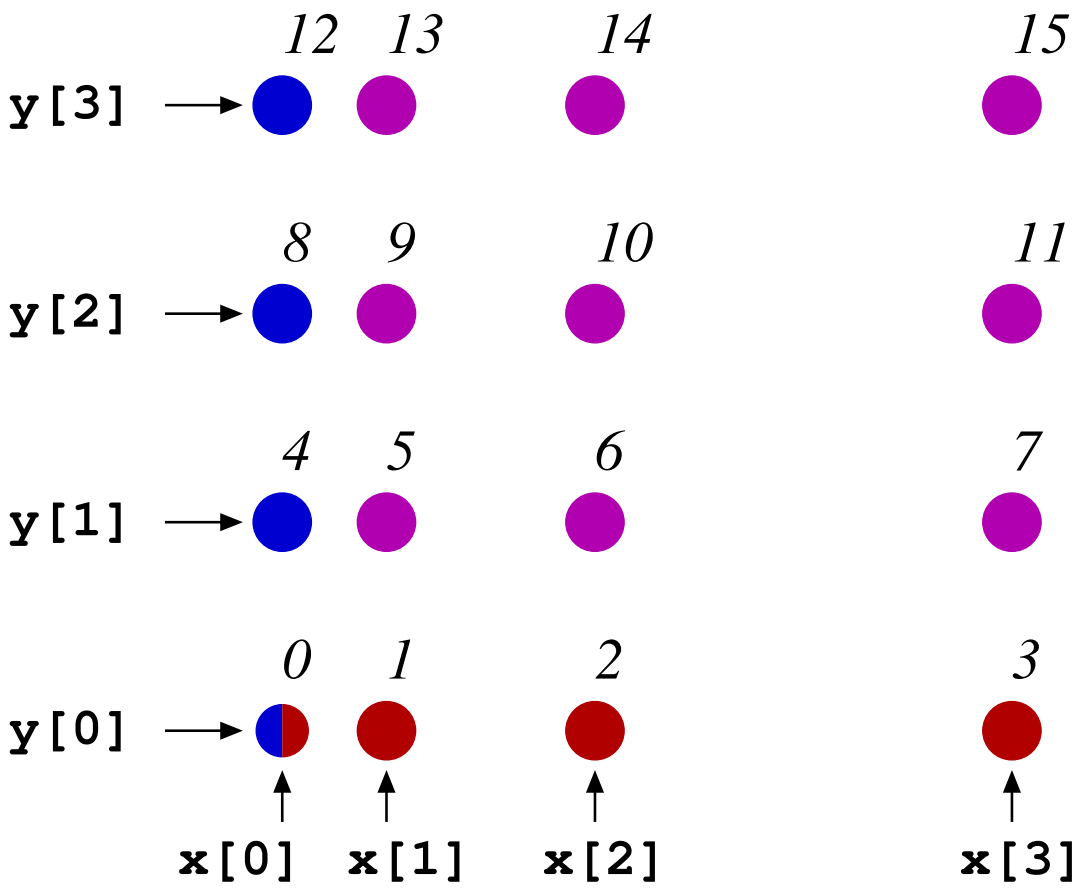
```
#include <vector>
#include <utility>
class Grid
{
  std::vector<double> _x, _y;
public:
  typedef std::pair<double, double> Point;
  Point get_point(unsigned m)
    { return Point(_x[m], _y[m]); }
};
```

- The `get_point()` function returns an object of type `Point`
- Once again: should check input arguments!

- Suppose we need to use a special type of `Grid`
- The nodes of this special grid shall be formed by a *Cartesian product* of the `_x` and `_y` `vectors`
- The `CartesianGrid` will still require `2n` storage but will actually represent n^2 nodes



- Magenta nodes formed by Cartesian products of the red and blue nodes
- Red and blue nodes are stored in the `x` and `y` vectors of the `Grid` class



- Let there be `n` nodes in each direction
- Assume we number the nodes sequentially as shown
- The coordinates of node `m` are given by: (`x[m%n]`, `y[m/n]`)

For reference:

- The modulus operator, `m%n == m - n*floor(m/n)`,
- Integer division, `m/n == floor(m/n)`
- Where `floor(x)` ≡ "round to largest integral value not greater than `x`"

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- We need to take a few more steps to get the `Grid` class ready for subclassing . . .

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- To make the _x and _y data members accessible to subclasses, we must first declare them protected

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- The `protected` data members are still not accessible by external clients of the class

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- Every class which has subclasses should also have a virtual destructor

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- The `virtual` keyword is used to label any function which will be redefined by subclasses

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual ~Grid() {}
};
```

- When applied to destructors, the `virtual` keyword allows objects of type `Derived*` to be destroyed via objects of type `Base*`

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid : public Grid
{
public:
  CartesianGrid(unsigned n=0)
    : Grid(n) {}
};
```

- We can now define the `CartesianGrid` class

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid : public Grid
{
public:
  CartesianGrid(unsigned n=0)
    : Grid(n) {}
};
```

- To declare that one class is derived from another, we follow the name by a colon, the type of inheritance, and the name of the parent class

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid : public Grid
{
public:
  CartesianGrid(unsigned n=0)
    : Grid(n) {}
};
```

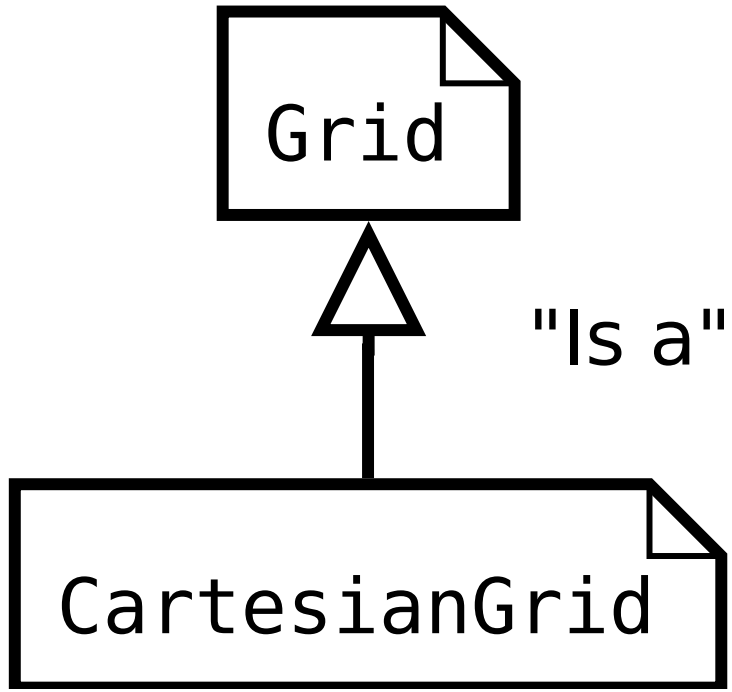
- The type of inheritance used here is `public`, which is the standard type used in polymorphism

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid : public Grid
{
public:
  CartesianGrid(unsigned n=0)
    : Grid(n) {}
};
```

- Derived classes also need constructors
- Often, as here, we can simply call the base class constructor

- Inheritance implements an "Is a" relationship between objects
- We say "`CartesianGrid` is a `Grid`"



grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual unsigned n_points()
  { return n(); }
};
```

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid
  : public Grid
{

public:
  virtual unsigned n_points()
  { return n() * n(); }
};
```

- One key difference between the two types of `Grid` is the number of actual points they represent

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual unsigned n_points()
  { return n(); }
};
```

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid
  : public Grid
{

public:
  virtual unsigned n_points()
  { return n() * n(); }
};
```

- We can characterize this by defining a `virtual n_points()` function

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual unsigned n_points()
  { return n(); }
};
```

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid
  : public Grid
{

public:
  virtual unsigned n_points()
  { return n() * n(); }
};
```

- In the base class, `n_points()` simply returns `n()`, the size of the data vectors `_x` and `_y`

grid.h

```
#include <vector>

class Grid
{
protected:
  std::vector<double> _x, _y;
public:
  virtual unsigned n_points()
  { return n(); }
};
```

cartesian_grid.h

```
#include "grid.h"

class CartesianGrid
  : public Grid
{

public:
  virtual unsigned n_points()
  { return n() * n(); }
};
```

- In the derived class, `n_points()` returns `n()*n()`, the number of points represented in the Cartesian product `Grid`

grid.h

```
class Grid
{
...
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m],_y[m]);
  }
};
```

cartesian_grid.h

```
class CartesianGrid
  : public Grid
{
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m % n()],
                _y[m / n()]);
  }
};
```

- The manner in which the points are accessed is also different between the two `Grids`

grid.h

```
class Grid
{
...
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m],_y[m]);
  }
};
```

cartesian_grid.h

```
class CartesianGrid
  : public Grid
{
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m % n()],
                _y[m / n()]);
  }
};
```

- We can characterize this by making `get_point()` `virtual` in the base class

grid.h

```
class Grid
{
...
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m],_y[m]);
  }
};
```

cartesian_grid.h

```
class CartesianGrid
  : public Grid
{
public:
  virtual Point
  get_point(unsigned m)
  {
   return Point(_x[m % n()],
                _y[m / n()]);
  }
};
```

- And then redefining it in the derived class

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- We are now ready to see how inheritance helps enable code reuse

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- We first define the `f` function which takes a reference to a `Point` and computes the scalar function value $f(x, y) = xy$

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- The const keyword is required so that we can use this function with an unnamed temporary

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- `first` and `second` are data members of the `std::pair` class, here they correspond to `x` and `y` values

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- The `grid_function()` takes a reference to a `Grid` object, hence it can be called with a derived type as well

main.C

```
#include "cartesian_grid.h"

double f(const Grid::Point& p)
{ return (p.first * p.second); }

void grid_function(Grid& g, std::vector<double>& v) {
  v.resize(g.n_points());

  for (unsigned i=0; i<g.n_points(); ++i)
    v[i] = f(g.get_point(i));
}
// Contd. on next slide
```

- The `grid_function()` uses the virtual `n_points()` and `get_point()` functions

```
// Contd. from previous slide

int main() {
  Grid g;            // Assume g, cg are somehow
  CartesianGrid cg; // filled with values...
  std::vector<double> v;

  grid_function(g, v);
  grid_function(cg, v);

  return 0;
}
```

- Example code calling the `grid_function()`

```
// Contd. from previous slide

int main() {
  Grid g;              // Assume g, cg are somehow
  CartesianGrid cg; // filled with values...
  std::vector<double> v;

  grid_function(g, v);
  grid_function(cg, v);

  return 0;
}
```

- The *same function* can be called on either type of grid to compute grid function values at the nodes

```
// Contd. from previous slide

int main() {
  Grid g;             // Assume g, cg are somehow
  CartesianGrid cg; // filled with values...
  std::vector<double> v;

  grid_function(g, v);
  grid_function(cg, v);

  return 0;
}
```

- `grid_function()` need not be rewritten if a new type of `Grid` (supporting the same interface) is developed

1. Recall the previous matrix example which utilized a

```
vector<vector<double> > matrix;
```

   Write code to define a `Matrix` class. This code should make several of the operations you coded in the previous exercise easier to use and reuse.
2. As the designer of the `Matrix` class, treat other codes (even your own) which might use the `Matrix` as "clients". What interfaces would clients of a `Matrix` class expect to have?

3. "Polymorphic collections" of objects are possible through `vectors` of pointer-to-Base, or `vector<Base*>`.
   a.) Write code to implement a hierarchy of 2D geometric shapes including `Circle`, `Square`, `Trapezoid`, etc., with a common base, `Shape`.
   b.) Define a virtual `area()` function which can be suitably overloaded in subclasses.
   c.) Create a collection of shapes, for example,

```
Circle c(0.5); Square s(1.0);
vector<Shape*> vs(2);
vs[0] = &c; vs[1] = &s;
```

   and write code to loop over the collection, computing the area of each `Shape`.

4. Think about your personal application/research code at a high level, in terms of the key data structures and algorithms, and the way they interact with one another. Imagine a minimal, coarse-grained set of objects and interfaces which can encapsulate this interaction, possibly sketching a diagram showing the interrelationships.

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

Yes!

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

Yes!

- But, this is like asking if driving a car is 'dangerous'.

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

Yes!

- But, this is like asking if driving a car is 'dangerous'.

- It is dangerous, but it is also a very convenient and effective means of transportation.

- Wait, this is supposed to be "Scientific Software." Isn't OO code 'slow'?

Yes!

- But, this is like asking if driving a car is 'dangerous'.
- It is dangerous, but it is also a very convenient and effective means of transportation.
- As long as everyone plays by the rules, nobody gets hurt!

- Consider a simple example using a vector to implement row-major storage.

```
#include <vector>
...
int matrix_size = 10000;
std::vector<double> v(matrix_size*matrix_size);

int cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    v[i*matrix_size+j] = cnt++;
```

- Consider a simple example using a vector to implement row-major storage.

```
#include <vector>
...
int matrix_size = 10000;
std::vector<double> v(matrix_size*matrix_size);

int cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    v[i*matrix_size+j] = cnt++;
```

- We can instead hide the index calculation in a user-defined Matrix type:

matrix.h

```
#include <vector>

class Matrix
{
public:
  Matrix(int mm, int nn);
  double& operator()(int i, int j);
private:
  int m, n;
  std::vector<double> vals;
};
```

- The user code is now:

```
#include "matrix.h"
...
int matrix_size = 10000;
Matrix m(matrix_size, matrix_size);

int cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    m(i,j) = cnt++;
```

- Timing results (in seconds, averaged over 5 runs) for the two different versions with different optimization levels.

	–O0	**–O3**
std::vector	5.44	1.72
Matrix	6.10	1.70

- With a decent compiler (in this case, g++) there is almost no difference in performance between the two.
- Does not require much advanced optimization knowledge on the part of the user (e.g. expression templates).
- The "object" code is arguably cleaner, and provides better reuse possibilities.

- Virtual functions: an OO feature frequently cited as "slow."
- Modify previous Matrix class to allow subclassing:

```
class MatrixBase
{
public:
  MatrixBase(int mm, int nn);
  virtual ~MatrixBase() {}
  virtual double& operator()(int i, int j) = 0;
protected:
  int m, n;
  std::vector<double> vals;
};
```

- Define the MatrixRowMajor subclass to implement row-major storage:

```
class MatrixRowMajor : public MatrixBase
{
public:
  MatrixRowMajor(int mm, int nn);
  virtual double& operator()(int i, int j);
};

double& MatrixRowMajor::operator()(int i, int j)
{
  return vals[i*n + j]; // row major
}
```

- Define the MatrixColMajor subclass for column-major storage:

```
class MatrixColMajor : public MatrixBase
{
public:
  MatrixColMajor(int mm, int nn);
  virtual double& operator()(int i, int j);
};

double& MatrixColMajor::operator()(int i, int j)
{
  return vals[i + m*j]; // col major
}
```

- (Essentially) the same matrix-fill code can be re-used...

```
MatrixRowMajor m(matrix_size, matrix_size);

int cnt=0;
for (int i=0; i<matrix_size; ++i)
  for (int j=0; j<matrix_size; ++j)
    m(i,j) = cnt++;
```

- Average timing results (in seconds) for the original and polymorphic Matrix classes.

	–O0	**–O3**
Matrix	6.10	1.70
Matrix (virtual, row-major)	6.08	1.98
Matrix (virtual, col-major)	8.06	3.77

- The additional flexibility obtained by decoupling the storage layout from the algorithm cost us about 15% in the row-major case.

- Also, the "generic" algorithm (which is inherently row-major) did not perform nearly as well on the column-major layout.

- We can address both these issues by becoming virtual at a "higher level."

- Recognizing that the algorithm is not efficiently decoupled from the storage layout, we can make the *algorithm itself* virtual instead.

```
class MatrixBase
{
public:
  MatrixBase(int mm, int nn);
  virtual ~MatrixBase() {}
  virtual void fill() = 0;
protected:
  int m, n;
  std::vector<double> vals;
};
```

- Implemented for the row-major case (col-major case is analogous):

```
void MatrixRowMajor::fill()
{
  int cnt=0;
  for (int i=0; i<m; ++i)
    for (int j=0; j<n; ++j)
      vals[i*n + j] = cnt++;
}
```

- And finally, called generically from user code:

```
MatrixRowMajor m(matrix_size, matrix_size);
m.fill();

// Or...

MatrixColMajor m(matrix_size, matrix_size);
m.fill();
```

- Combined results for the original, non-virtual objects and the virtual `fill()` function.

	–O0	**–O3**
std::vector	5.44	1.72
Matrix	6.10	1.70
fill(), row-major	5.70	1.68
fill(), col-major	5.71	1.69

- Proper use of virtual functions (i.e. not too many) leads to more flexible code with the same performance as less flexible code.
- The `fill()` method in this example can be made more sophisticated if we also pass a "`Filler`" function object to it.
- This example was trivial: there are libraries (boost/blitz/eigen) which are much more realistic.
- The guidelines developed here for using virtual functions should apply in other situations as well.

1. Flesh out the example in this section by writing an actual code using the snippets here for guidance. Compare/verify the reported timings for the different cases on your system.

2. Extend the example from this section by adding the pure virtual function

```
virtual void matvec(vector<double>& x,
                    vector<double>& b) = 0;
```

to the `MatrixBase` class, which computes the matrix-vector product $Ax = b$.

a.) Add specializations for row- and column-major subclasses
b.) Compare timings for your specialization with hand-coded loops using `vector<double>`

3. Operator overloading is a feature of C++ which provides "syntatic sugar", that is, makes code look prettier.

   a.) Fill in the body of the following free function which overloads the '`*`' operator. *Hint: reuse the* `matvec()` *function from the previous exercise!*

```
vector<double> operator*(MatrixBase& A,
                         vector<double>& x);
```

   b.) Demonstrate the use of your function by writing code like

```
b = A*x;
```

   for suitably-defined `b`, `A`, and `x`.

- Introduction
- Data Types
- Control Structures
- I/O with Streams
- Pointers, References, Arrays, and Vectors
- Objects
- C++ in Scientific Software
- Additional References

- IOStreams and General Reference
  `http://www.cplusplus.com`
- STL reference
  `http://www.sgi.com/tech/stl/table_of_contents.html`
- Frequently Asked Questions (Esoteric)
  `http://www.parashift.com/c++-faq-lite`
- C++ Newsgroup ("language lawyers")
  `groups.google.com/group/comp.lang.c++.moderated`

- "Thinking in C++," Vols. 1 and 2 by Bruce Eckels (google: "thinking in c++ pdf")

- All of the "Effective ..." books by Scott Meyers:
  - Effective C++
  - Effective STL
  - More Effective C++

- Herb Sutter, `http://herbsutter.com/`
- Dave Abrahams, `http://daveabrahams.com/`
- Andrei Alexandrescu, `http://erdani.com/`
- Pete Becker, `http://www.petebecker.com/`
- Jaakko Järvi, `http://parasol.tamu.edu/~jarvi/`
- Nicolai Josuttis, `http://www.josuttis.com/`
- Peter Dimov, `http://www.pdimov.com/`
- Doug Gregor, `http://www.osl.iu.edu/~dgregor/`