C, ++?

Bálint Joó (bjoo@jlab.org)
Jefferson Lab, Newport News, VA

given at

Jefferson Lab Graduate Lecture Series

July 5, 2006

Hello World

- File: hello.cc

```
#include<iostream>

using namespace std;

int main(int argc, char *argv[])
{
    cout << “Hello World” << endl;
    return 0;
}
```

- Make it run:
  - Compile: **g++ -o hello hello.cc**
  - Run: **./hello**
  - Output: **Hello World**

What does it all mean

- int main(int argc, char *argv[]) { ... }
  - signals this is the main program (PROGRAM)
  - program goes between { } (BEGIN/END)
- statements punctuated by semicolons (;)
- cout << "Hello World" << endl;
  - Write "Hello World" and newline to std output
- return 0;
  - Return an error code of 0 to indicate success

Compiling

- g++ -o hello hello.cc
  - .cc file is the C++ source
  - -o hello means put compiled program into 'hello'
  - this step
    - turns hello.cc into object file (Compilation)
    - turns object file into executable (Linking)
    - intermediate object file is removed
  - More about this process much later.

Variables & Basic Types

- Variables: Things we can assign values to and manipulate by name (eg: print them)

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

int main(int argc, char *argv[])
{
    // Signed Integer (FORTRAN: INTEGER)
    int fred=5;
    cout << “Fred has value “ << fred << endl;

    // Single Precision Floating Point (FORTRAN: REAL)
    float george = 2.5;
    cout << “George has value “ << george << endl;

    // Double Prec. (FORTRAN: DOUBLE PRECISION REAL)
    double jim=2.5; // Double precision floating point value
    cout << “Jim has value ” << jim << endl;

    // C++ String
    string mike = “A string of text”;
    cout << “Mike has value “ << mike << endl;
}
```

Syntactic notes

- Comments (for your own info. Ignored by compiler)
  - Single line:

    ```
    // All characters following slashes ar comment
    ```

  - Multi line

    ```
    /* Comment starts here and goes on
       onto the next line */
    ```

- Declare and Define Variables where you need them
  - No need to declare all up front like FORTRAN/C
  - Helps program readability

Basic Types

- Integer types: eg int
  - Signed: char, short, int, long, long long
  - Unsigned: unsigned char, unsigned short, etc
- Numeric types: eg float, double
  - No native Complex type like FORTRAN
- Boolean Logic values
  - bool (can have values true or false)
- Pointers/References – get onto these later

Does size matter?

- Chief difference between char, int, etc:
  - size in memory, therefore numerical range
  - On 32 bit architectures typical sizes are
    - unsigned char : 1 byte, represent an ASCII character
    - unsigned short : 2 bytes (0 – 64K)
    - unsigned int : 4 bytes (0 - 4G)
    - unsigned long : 4 bytes (represent memory word)
    - float: 4 bytes (resolve up to 6-7 digits)
    - double: 8 bytes (resolve up to 14 or so digits)
  - Signed integer types: lose 1 bit for sign

Things to do with variables

- Assignment / Re assignment (= operator)

```
int i = 5;

int j = 6;

int i_again = i; // i_again is assigned value of i

i  = 4; // old value is now forgotten, i_again is still 5
```

- Expressions

```
int sum = i + j;  // sum is assigned value of 11

int product = i * j; // product is 30

int int_quotient = i / j ; // int_quotient is 0
```

Integer division and remainder

- Integer division may be strange at first

```
int i = 12;

int j= 8;

int quotient = i / j; // quotient = 1

int remainder = i % j; // remainder = 4
```

- Floating point division is more like your calculator

```
float f = 12;

float h = 4;

float float_quotient = f / h; // Qoutient is 1.5
```

Logical values and operations

- Boolean logic values represented by bool-s
  - bool true_value = true;
  - bool false_value = false;
- Conditional expression:
  - Equality (==) : bool are_a_and_b_equal = (a == b);
  - NOT (!) : bool not_true = (! true);
  - Inequality: bool are_a_and_b_different= (a != b);
  - AND (&&): bool a_and_b = (a && b)
  - OR (||): bool a_or_b = (a || b)

Loops I

- While Loops (DO WHILE)

```
while( conditional expression ) {
    // Body  between the curly braces
}
```

- If conditional is false at start, body never executed
- Example:

```
int i = 0;

while(i < 10) {
    i+=1; // Increment i by 1.
}

cout << i << endl; // should print 10
```

Loops II:

- Do loops: (REPEAT UNTIL)

```
do {
    // Loop body
} while (conditional);
```

- Similar to while-loop but Loop Body is guaranteed to be executed at least once

```
int input = 0;
do {
    cout << “Enter the number a number greater than 5” << endl;
    cin >> input;          // Read an integer
} while ( input <=  5 );
```

Loops III

- For loops (DO i=...)

  for(initial setup; continue condition; end action) {

  // Body

  }

- Typical use:

```
for( int = 0;   i < 6; i++) {
    // Start with i=0, repeat the loop while i < 6,
    // increment i at the end  of body.

    // i does not exist outside loop

    cout << i;
}
```

For loops

- for loop is equivalent to specialised while loop

```
{ // Scope unit so i doesn't go outside the loop
    int i = 0;
    while(i < 5) {
       cout << i << endl;
       i++;
       }
    //  i goes out of scope (disappears) here
}
```

- Is exactly equivalent to:

```
for( int i = 0; i < 5 ; i++) {
   cout << i << endl;
}
```

- Can use more complex initialisations/conditions/end of loop actions. For loop is a syntactic convenience

Conditional Execution

- Either or actions (depending on condition)

```
if ( condition ) {

  // Body to execute if condition is true

}
```

- Example:

```
int i;
cin >> i ;   // Read the integer

if ( i > 0 ) {
    cout << " You have entered a positive integer" << endl;
}
```

Conditional Execution II

- If - Else

```
if( condition ){
  // Body if condition is true
}
else {
  //  Body if condition is false
}
```

- Example

```
int i;
cout << "Enter an integer " << endl;
cin >> i ;

if ( i < 0 ) {
    cout << " Negative value entered " << endl;
}
else {
    cout << " Nonnegative value entered" << endl;
}
```

Conditional Expression IV

- If-else if-else
- Example:

```
int i;
cout << “Enter an integer “ << endl;
cin >> i;

if( i == 0 ) {
    cout << “ You entered zero “ << endl;
}
else if ( i < 0 ) {
    cout << “ You entered a negative number “ << endl;
}
else {
    cout << “ You entered a positive number “ << endl;
}
```

Switch statements (Case statements)

- Select between a set of enumerated cases:

```
int i = 0;
cout << “Enter a number” << endl;
cin >> i;

switch ( i )  {
  case 0:  cout << “You entered 0” << endl;
           break;

  case 1:  cout << “You entered 1” << endl;
           break;

  default:  cout << “You entered something else other than (0 and 1)”
                 << endl;
           break;
}
```

- can combine cases by omitting 'break'
- execution falls through to next case (DANGEROUS)

Case Statements II

- Rope to hang yourself with:

```
int i;
cout << “Enter a number “ << endl;
cin >> i;

switch( i ) {
  case 0: // No 'break'

  case 1: // No 'break'

  case 2: cout << “You have entered 0 or 1 or 2” << endl;
          break;

  default: cout << “You have entered something else << endl;
           break;
}
```

- Common mistake: forgetting the break before default

Case Statements: Final Word

- Semantics of case statement are complicated
- Easy to screw up
- C++ obfuscation: declaring variables in one 'case' has effect on 'other' – use {} to set scope
- Best to avoid using case at all – there are other, better methods

Functions

- Collect frequently used statements into a function
- C/C++ equivalent of Fortran FUNCTION and SUBROUTINE.
- Has one single return value
  - this can be "void" - to mean no return value
- First example of
  - Code modularisation (factoring)
  - Code re-use

Examples:

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

void printUsage(void)
{
   cout << " Type: ./myprog " << endl;
   return;
}

int sumInt(int from, int to)
{
   int ret_val=0;
   if( from == to ) {
     return  from;
   }
   else {
      int start=0, end=0;
      if( from < to ) {
       start=from; end = to;
      }
     else {
          start=to; end=from;
     }
     for(int i=start; i <=end; i++) {
         ret_val += i;
     }
   }
   return ret_val;
}
```

```
int main(int argc, char *argv[])
{
   if ( argc != 1 ) {
    printUsage();
    return 1;
   }

   int start=4, end=6;
   int sum = sumInt(start, end);

   cout << "Sum is " << sum << endl;

   start=6;
   sum = sumInt(start, end);

   cout << "Sum is " << sum << endl;
   return 0;
}
```

Argument Passing

- C/C++ - standard way: "pass by value":
  - void foo(int a, float b)
    - foo() gets **COPIES** of a and b
    - changing a and b inside foo() does not change them outside foo.
- C++ addition: "pass by reference"
  - void foo2(int& a, float& b)
    - int& is a C++ type called an **integer reference**
    - changing a and b in foo2() also changes **the variable they refer to** outside foo2

References

- A C++ reference type is an 'alias' to something. A kind of handle if you will. You can treat it as the thing which it is referencing:

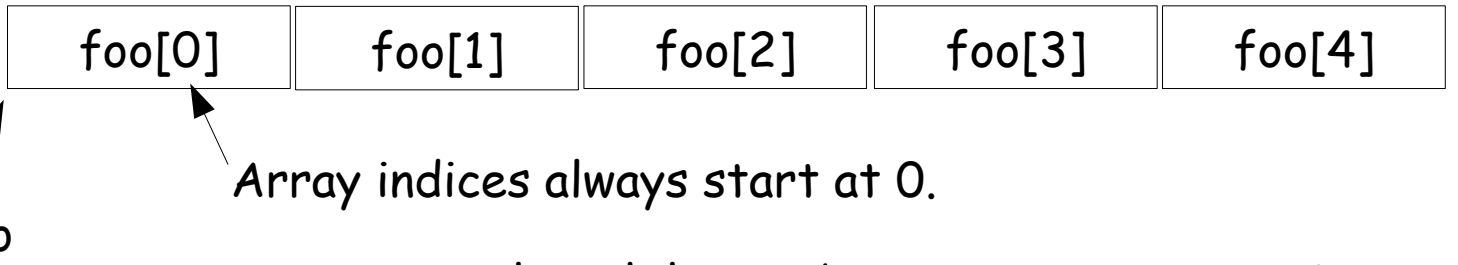
```
#include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
    int x = 5;          // The original
    int& x_ref = x;     // x_ref is a reference to x
    x_ref++;            // Increment the variable to which x_ref refers

    cout << “x is now ” << x << endl;    // Should print x is now 6
    cout << “x_ref is  “ << x_ref << endl;  // Should print x_ref is 6

    return 0;
}
```

C Style Arrays

- C only really supports 1D arrays
- declare array using []:

  float foo[5];

  - declares a contiguous array with 5 elements



Array length has to be a constant expression (not a variable)

Eg:

```
float foo[5];
for(int i=0; i < 5; i++) {
  foo[i] = 6;
}
```

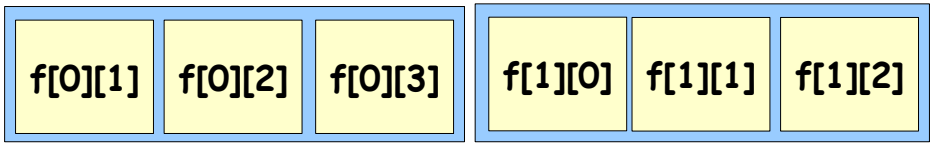
Sets every element to 6

Extra Array Dimensions

- A 2D array is notionally an array of 1D arrays

float f[2][3];

Array of 2 elements, each of which is an array of 3 floats

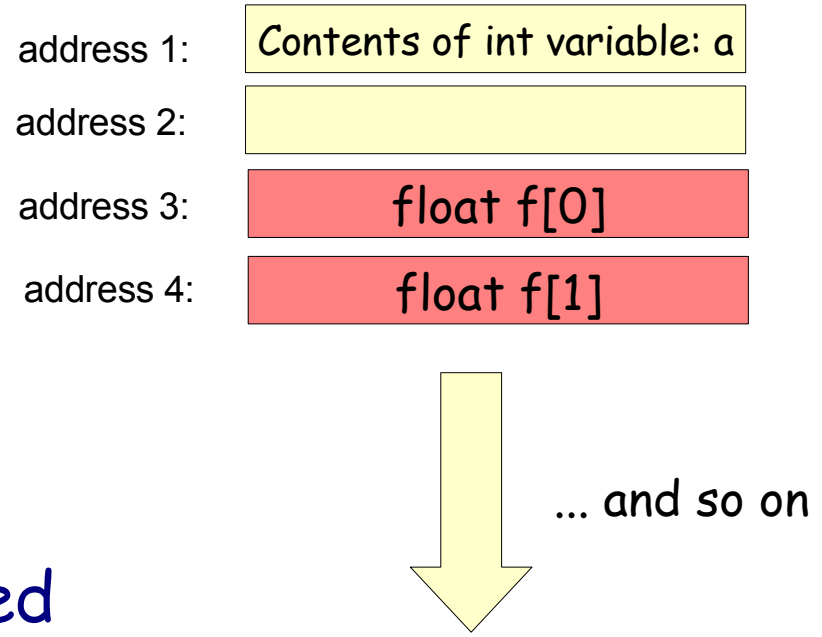


- Rightmost index runs fastest (Row Major Order)
  - Opposite to FORTRAN
- Can Slice ie refer to just f[0]
- Arrays declared like above are contiguous in memory

Pointers

- Data items take space in memory
- Think of computer memory like an array. Each item of data occupies some elements of this array.
- Each datum has a location also known as an address

Example: a has address 1
f[0] has address 3
f[1] has address 4



- A special type of variable that holds an address is called a pointer variable

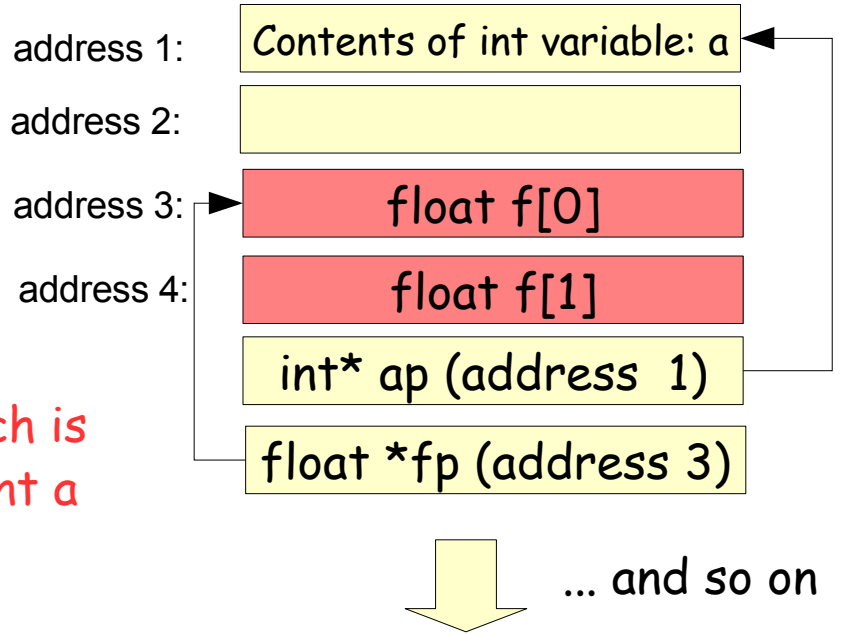
Setting Pointers with the address operator

- A pointer to a type is denoted by the type followed by an *
  - pointer to an int is denoted int*
  - pointer to a float is denoted float *

- Address of a variable: & operator

```
int* a_ptr = &a;
```

means: set the contents of a_ptr (which is an integer pointer) to the address of int a
or: make a_ptr point to a



De-referencing pointers

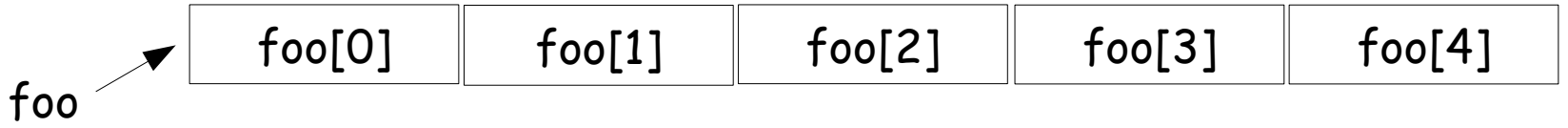
- We can get at the value at the end of the pointer by using the * operator;

```
int* a_ptr = &a;        // Set a_ptr to the address of a

// Set a_copy to the value at the end of a_ptr

int a_copy = *a_ptr;

(*a_ptr)++;        // Increment value at end of a_ptr
```

- NULL pointer (Pointing to nothing)
  - Special value: 0 in C++
  - De-referencing a NULL pointer is an error.

Arrays and Pointers II

- Array is just a pointer



  - int foo[5];
  - int* f = foo; // f now points to foo[0]
  - int foo2D[5][3];
  - int** f2D = foo2D; // f2D points to foo2D[0]
  - and foo2D[0] is just a pointer to foo2D[0][0]
  - *f = 5; // Sets f[0] = 5
  - *(f + 1) = 6; // Sets f[1] = 6

Passing by Pointer to a function.

- We have seen 'pass by value' to a function
  - give a COPY of the value to the function
- We have seen 'pass by reference' to a function
  - give a COPY of a REFERENCE
- We can also 'pass by pointer'
  - eg to pass an array:
  - void foo(int *f); // A function that takes an int *
- But function won't know how big the array is

Example: Square Norm of a vector

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

double norm2(double *vector, int length)
{
   if (vector == 0) {
     cerr << "NULL Pointer argument" << endl;
     exit(1);
   }
   double ret_val = vector[0]*vector[0];
   for(int i=1; i < length; i++) {
     ret_val += vector[i]*vector[i];
   }
   return ret_val;
}

int main(int argc, char* argv[])
{
    double vec3[3] = { 3.2, 4.1, 5.6};
    double n = norm2(vec3, 3);
    cout << "|| vec3 || = " << n << endl;
}
```

Expecting an array (double *), with length in 'length'

Sanity check. Make sure vector is not a NULL pointer

Could have written (*vector)*(*vector) but would be harder to read

Static Array Initialisation

Function Call

Message:

C and C++ always pass native arrays by pointer. Native arrays ARE pointers

Congratulations

You can now program Fortran (77)
in C++

Grouping Data, Data Structures

- Previous example. Would be nice group together the array and size. This can be done using C structs
  - struct MyVector {

    double vector*;

    int length

    };
  - At this point the pointer is uninitialised.
  - Q: What does it point to? Where is the array?
  - A: Nowhere! You don't want to mess with it yet.

Initialising the Struct

- Can write a function to do this:

```
void initVector(MyVector& vec, int length)
{
   vec.vector = new int [ length ];
   if( vec.vector == 0 ) {
      cerr << "Couldnt allocate vector" << endl; exit(1);
   }
   vec.length = length;
}
```

Tries to grab space for a vector of length integers. Returns pointer to space or 0 if unsuccessful

The . lets you refer to parts of a struct

So What's new ?

- Pointer can be initialised using the 'new' keyword
  - Allocates memory for the object and returns a pointer to its location.
  - new is typed (as opposed to C's malloc)
    - float *f = new float; // returns float *
    - int *f = new int; // returns int *
  - array allocated with ' new [] '
    - int *int_array = new int [10];
  - new returns 0 poiner or throws exception if requested memory is not available

Delete

- Automatic variables are cleaned up when you reach the end of the variable scope

```
{ // Start new scope unit
      int x=5;
} // Scope unit is ended. x disappears, its memory is
  // reclaimed
```

Automatic variable created and initialised

- Memory allocated with new does not get cleaned up automatically. You must explictly free it using delete
  - int *i = new int; delete i;
  - int *i = new int [10] ; delete [] i;

regular delete

array delete

Memory Leaks (not delete-ing after new)

- Try the following:

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

int main(int argc, char *argv[])
{
    int want = 2;
    int *pointer;
    while (true) {
            cout << “Want is  “ << want << endl;
        pointer = new int [ want ]; // No Delete []
        want *= 2;
    }
}
```

- On my laptop I reached want=536870912
- then the program crashed – it was out of memory
- This is called a memory leak. An insiduous bug

Back to our vector

- We used new in initVector()
- So we need a destroyVector() to clean up or we suffer a memory leak.

```
void destroyVector( MyVector& vec )
{
    if( vec.vector != 0 ) {
        delete [] vec.vector;
    }
    vec.length = 0;
}
```

- We can think of other things to do with our vector:
  - set all elements to zero, copy the contents of other vectors into mine, change vector size etc.

Danger!!!

- Resize operation is quite complex:
  - need to set new length
    - can do this by assigning new length to vec.length
  - But it is NOT enough – array must be resized too
  - But the user has NO PROTECTION (he or she is free to erroneously reset length)
  - Would be better to
    - not let the user change length explicitly
    - provide a function to resize correctly.

Towards Object Orientation

- Objets – Beyond simple structures
  - Group functions (initVector, destroyVector, copy, resize etc) together with the data (Encapsulation)
  - Hide the actual data so user doesn't manipulate it erroneously (Information Hiding)
  - There are two other Object Orientation Aspect
    - Inheritance (shared behaviours)
    - Polymorphism (treating different objects without regard to their type) – We'll get onto these later.
  - Look at the first two now

The MyVector Class (First C++ Attempt)

```
class MyVector {
private:
    double *vector;
    int length;
public:

  // Constructor (initFunction)
  MyVector(int size) : vector( new double [size] ), length(size) {}

  // Destructor (clean up function )
  ~MyVector(){ delete [] vector; length=0; }

  // Want to know length of vector for loops, but can't touch it
  // because it is now private. Here I return a copy.
  int getLength(void) const { return length; }

  // Array indexing - so I can treat vector like an array
  // This allows me to change the value in the vector (LHS of =)
  double& operator[]( int i ) { return vector[i]; }

  // Array indexing - this is read only access (RHS of =)
  const double& operator[]( int i )  const { return vector[i]; }

};
```

The internal parts (member fields) are now private. Only functions in the class can touch them. These constitute the state of MyVector

returns a pointer

One way to Initialise Member Fields

const double& return type means: You can't change the value of the reference returned number returned is read only

const before function body means: this function will not change my state (it won't modify internal fields)

Using MyVector

- We can now do some things with our vector:

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

#include “myVector.h” // Put the myVector code into file myVector.h
                      // We include the definition here

int main(int argc, char *argv[] )
{
    MyVector newVec(3);    // A vector of length 3 is created

    // Print the length of it – access member function using . like
    // a struct data member
    cout <<”length is:  “ << newVec.getLength() << endl;

    // assign some values (using read/write indexing method)
    newVec[0] = 1.5; newVec[1] = 2.0; newVec[2] = 3.0;

    // Print out its elements
    for(int i=0; i < newVec.getLength(); i++) {
       cout << “Vec[“<< i <<”] =” << newVec[i]; // Read Only []
    }
    // MyVector goes out of scope. Destructor is called. Memory is
    // cleaned up
}
```

What have we done

- Encapsulation of data and manipulation
  - collected manipulation functions and data
- Information Hiding
  - We have successfully hid the internal information
    - Private fields for the actual pointer and length
  - Protection
    - Array always allocated on creation – should not be null
    - I can't independently change the 'length' from outside the class

What else have we done

- Defined my own meaning for the [] operator
  - This is called 'operator overloading'
    - I can also overload =, *, +, - etc
  - This is an aspect of something called polymorphism
    - my arrays 'look and feel' like native arrays
- Terminology:
  - MyVector is a **class** - A definition of behaviour and data storage
  - newVector is an **object** - A concrete instance of the MyVector definition

Classes & Objects: A standard example

- Some more additions to the MyVector class
  - (can add these after the 'public:' in the class def'n)

```
void resize(int n)
{
    // destructive resize – array contents will be lost
    delete [] vector;              // Delete old
    vector = new double [ n ];     // Allocate new
    if( vector == 0 ) {
     cerr << “Failed to allocate vector in resize” << endl;
     exit(1);  // Allocation failure
    }
    length = n;
}

double norm2(void)  // Compute square norm
{
    double sum=0;
    for(int i=0; i < length; i++) {
         sum += vector[i]*vector[i];
    }
     return sum;
}
```

Using the new features

```
int main(int argc, char *argv)
{
    MyVector vec(2);
    vec[0] = 1.0;
    vec[1] = 2.0;

    cout << “Vector has square norm : “ << vec.norm2() << endl;

    vec.resize(3); // Now change it to have length 3;
    vec[0] = 1.0;
    vec[1] = 2.0;
    vec[2] = 3.0;

    cout << “Vector has square norm : “ << vec.norm2() << endl;
}
```

Checking Array Bounds

- Suppose I want to check that the user of MyVector doesn't try to get an element beyond the length of my vector. I could define a new class, which is identical to my old class (with a different name of course) and the indexing methods have safety checks

```
class MyCheckingVector {
private:
     int length;
     double *vector;
pubic:
     MyCheckingVector(int size) : vector (new double[size]),length(size) {}
     // All the rest: destructore and so forth

     double& operator[](int i) {
       if( i < 0 || i >= length ) {
          cerr << “index out of range error” << endl;
         exit(1);
       }
       return vector[i];
     }
};
```

Inheritance

- MyCheckingVector duplicates MyVector entirely, just to replace two member functions
  - Duplication !
  - Duplication is the Root of All Evil! Should be avoided!
  - C++ provides a different way: Inheritance:

```
class MyCheckingVector :  public MyVector {
      MyCheckingVector(int size) : MyVector(size) {} // Call MyVector's constructor
}
```

- This says: MyCheckingVector 'is a' MyVector and can access all MyVectors public fields/functions
- Constructor cannot be inherited. We provide it

Inheritance

- Here we say: MyCheckingVector inherits from MyVector OR
- MyVector is the Base Class of MyCheckingVector OR
- MyCheckingVector is a derived class of the Base Class of MyVector
- I can't inherit the constructor since the constructor is the 'name of the class'
- But I can call the constructor of the BaseClass
  - MyCheckingVector(int size) : MyVector(size) {}

Inheritance

- I can refer to MyCheckingVector as if it was a MyVector. In particular I can assign MyCheckingVector to a MyVector reference:

```
#include <iostream>
using namespace std;
#include "myvector.h"

int main(int argc, char *argv)
{
  MyCheckingVector newVec(3);

  MyVector& baseclass_ref = newVec; // Treat MyCheckingVector as a MyVector

  baseclass_ref[0] = 1.0;
  baseclass_ref[1] = 2;
  baseclass_ref[2] = 3;

  for(int i=0; i < baseclass_ref.getLength(); i++) {
    cout << "baseclass_ref["<<i<<"]= "<< baseclass_ref[i] << endl;
  }

}
```

Inheritance

- So far I have nothing new, just a different name for my vector. Now I can add its own operator[]:

```
class MyCheckingVector : public MyVector {
 public:
  MyCheckingVector(int size) : MyVector(size) {}

  // MyCheckingVector's own indexing function
  double& operator[](int i) {

    // Show we are calling this checked []
    cout << "Checking" << endl;
    if( i < 0 || i >= getLength() ) {
      cerr << "Index out of range error" << endl;
      exit(0);
    }
    // vector is private, I can't touch it but I can
    // call the public indexing method of the base class
    // if the check is OK
    return MyVector::operator[](i);
  }
};
```

Inheritance

- Almost successful. If I refer to MyCheckingVector explicitly everything works:

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

#include “myVector.h” // Put the myVector code into file myVector.h
                      // We include the definition here

int main(int argc, char *argv[] )
{
     MyCheckingVector newVec(3);    // A vector of length 3 is created

     // Print the length of it – access member function using . like
     // a struct data member
     cout <<”length is:  “ << newVec.getLength() << endl;

     // assign some values (using read/write indexing method)
     // Prints “Checking”
     newVec[0] = 1.5; newVec[1] = 2.0; newVec[2] = 3.0;

     // Print out its elements
     for(int i=0; i < newVec.getLength(); i++) {
         cout << “Vec[“<< i <<”] =” << newVec[i]; // Read Only []
     }
     // MyVector goes out of scope. Destructor is called. Memory is
     // cleaned up
}
```

Inheritance

- But if I refer to it as a MyVector, it doesn't print 'Checking'.

```
#include <iostream>
using namespace std;
#include "myvector.h"

int main(int argc, char *argv)
{
  MyCheckingVector newVec(3);

  MyVector& baseclass_ref = newVec; // Treat MyCheckingVector as a MyVector


  // Doesnt print 'Checking'
  baseclass_ref[0] = 1.0;
  baseclass_ref[1] = 2;
  baseclass_ref[2] = 3;

  for(int i=0; i < baseclass_ref.getLength(); i++) {
    cout << "baseclass_ref["<<i<<"]= "<< baseclass_ref[i] << endl;
  }

}
```

- Why Not?

Virtual Functions

- Because MyCheckingVector now has two versions of Read/Write operator[]:
  - One from MyVector()
    - called MyVector::operator[]
    - Used when referring to it as a MyVector&
    - Doesn't print 'Checking'
  - One from MyCheckingVector()
    - called MyCheckingVector::operator[]
    - Used when referring to it as MyCheckingVector
    - Does print checking

Virtual Functions

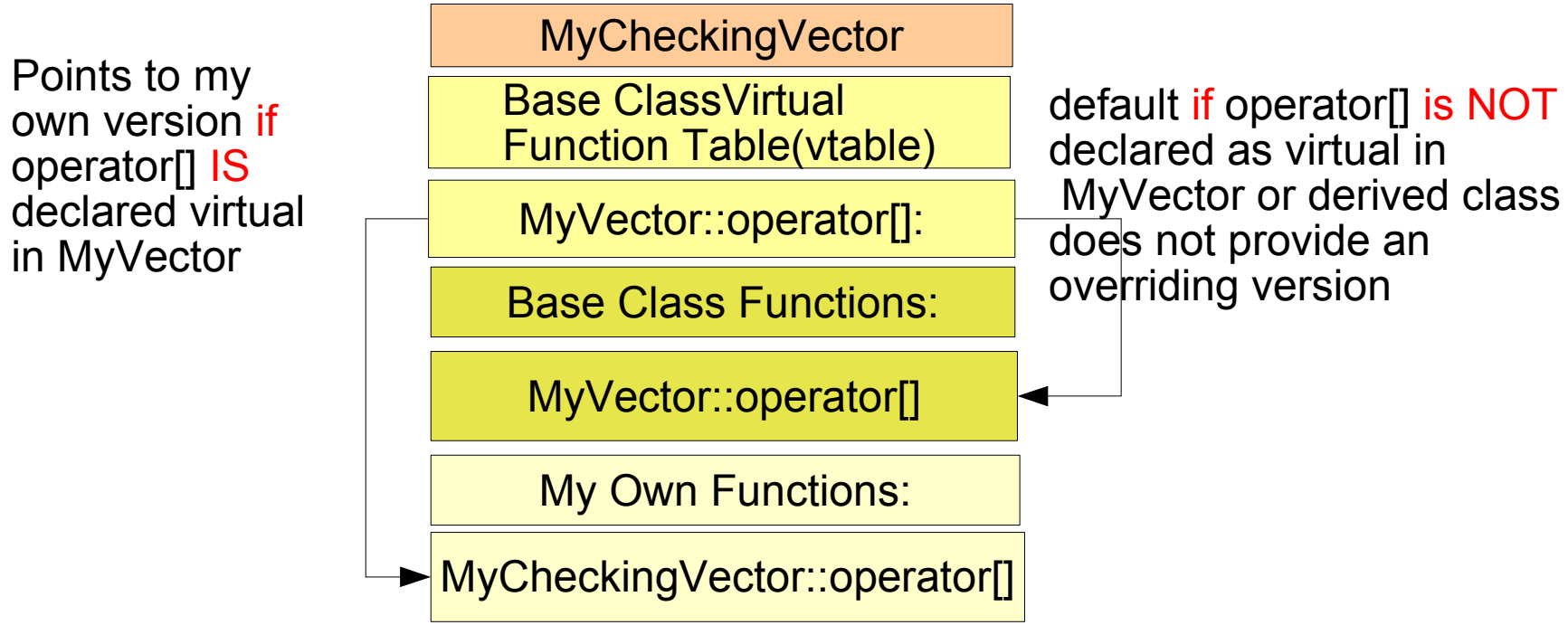
- For things to work properly we need to call MyCheckingVector::operator[] even when referring to the object as a MyVector&
- To do this we have to declare operator[] as 'virtual' in MyVector:

```
class MyVector {
  // ... omit for lack of space on slide
 public:
  MyVector(int size) : vector( new double[size] ), length(size) {
    cout << "MyVector::reating" << endl;
  }

  virtual double& operator[](int i) {
    return vector[i];
  }
```

Virtual Means: If a derived class overrides operator[] use the derived class's version

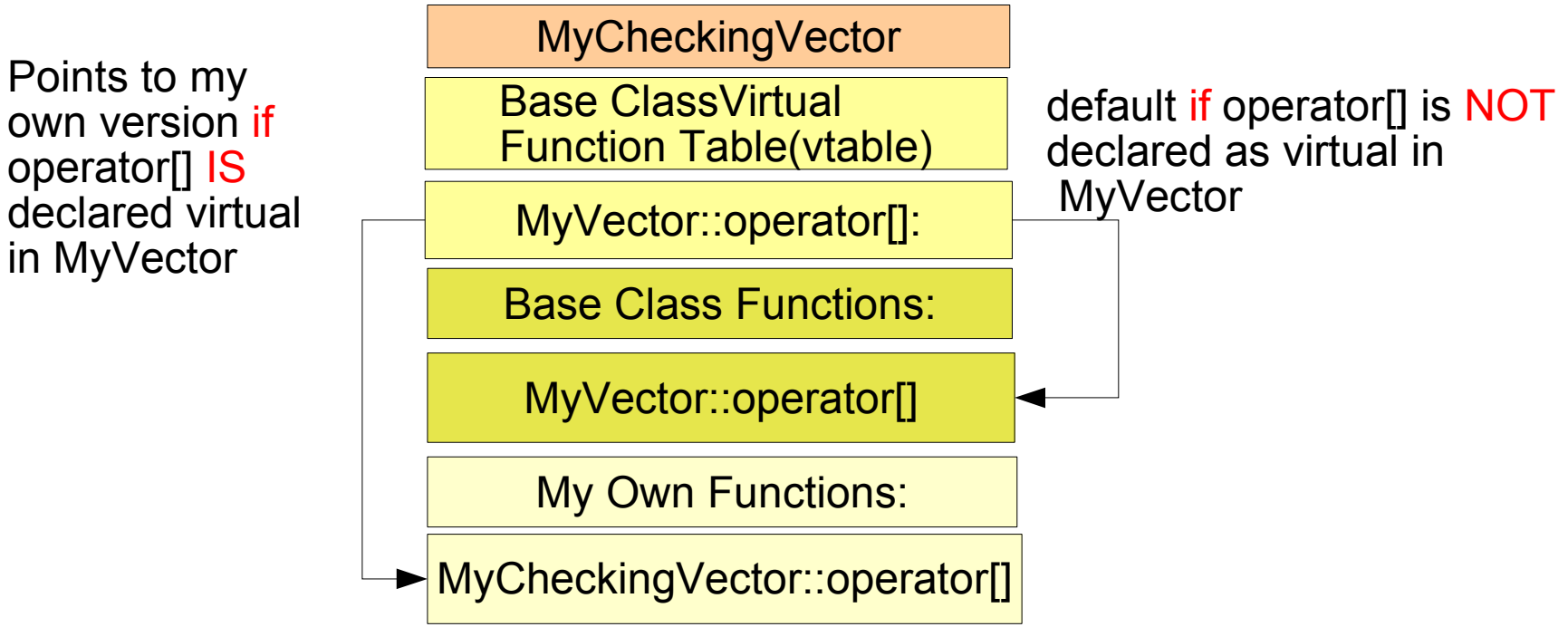
Virtual Functions

- With this change even referring to MyCheckingVector as a MyVector& prints 'Checking'
- How does this work?



Virtual Functions

- In the derived class is the 'vtable'. When referring to it through the base class (MyVector) the appropriate function is looked up in the vtable and called. Virtuality adds the cost of the indirection



Accessing data in the base class

- MyCheckingVector couldn't access 'vector' and 'length' in my vector because they are private and only the public functions are accessible in MyVector
- If I change MyVector to make 'vector' and 'length' protected instead of private MyCheckingVector can access them directly

```
class MyVector {
protected:  // Derived classes can
            // access these
    double *vector;
    int length;
public:
  // ... all the rest

};
```

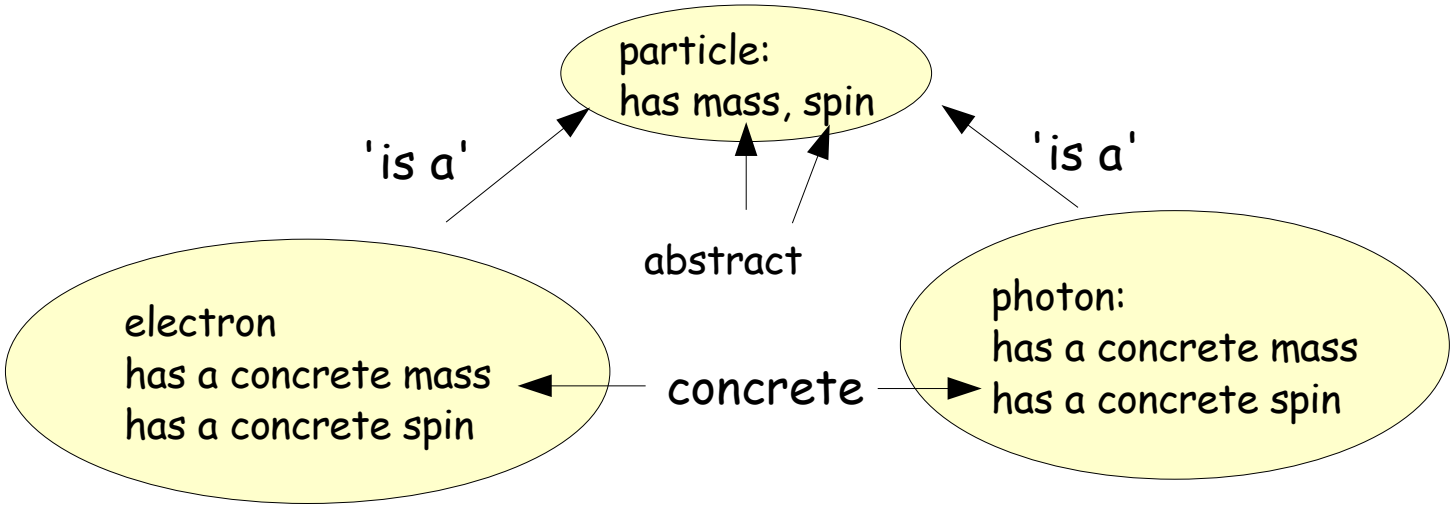
```
class MyCheckingVector: public MyVector{
public:
  double& operator[](int i) {
    if( i < 0 || i >= length ) {
      cerr << “Index out of range”
           << endl;
      exit(1);
    }
    return vector[i];
  }
}
```

Inheritance: Initial Summary

- Inheritance allows creation of hierarchies of related classes
- Polymorphism: Can use all derived classes as if they were the base class (Liskov Substitution Principle)
  - Need virtual functions to ensure the derived class's method is called even if it is refered to through the base class. Base class can provide default behaviour
- 'protected' members accessible to derived classes
  - 'private' members are only available in the class where they are declared.

Pure Virtuality (Abstract functions/classes)

- Often it does not make sense for the base class to provide a default. But want to define a function to specify an interface



- Both electrons and photons are particles and both have a concrete spin and mass. But we cannot correctly provide a default spin or mass for particle:

Pure Virtuality for Functions

```
class Particle {
public:
    // Pure Virtual Functions
    virtual double getSpin(void) const = 0;   // define but provide No
    virtual double getMass(void) const = 0;   // implementation or
                                              //  default:

    virtual ~Particle(); // Always call derived class's destructor
}
class Electron: public Particle {
public:
    // No data. C++ default constructor/destructor is OK
    double getSpin(void) const { return 0.5; } // spin 1/2
    double getMass(void) const { return 0.510998903; } // Mass in MeV
};

class Photon : public Particle {
public:
    // No data. C++ default constructor/destructor is OK
    double getMass(void) const { return 0; }  // Massless
    double getSpin(void) const { return 1; }  // Spin 1 boson
};
```

Particle has abstract members. Specifies an Interface

Electron and Photon specify the Implementation

Using Classes with Pure Virtual Members

- I cannot create an actual 'Particle' instance because some of its functions are abstract.
- But I can use a Particle& or a Particle* to manipulate Electrons and Photons.

```
void printDetails( const Particle& theParticle ) // Reference to Abstract
{
    cout << "Particle has mass : " << theParticle.getMass() << endl;
    cout << "Particle carries spin : " << theParticle.getSpin() << endl;
}

int main(int argc, char* argv[])
{
    Electron e;                 // Concreta
    Photon gamma;
    printDetails( e );          //pass Electron to printDetails as a Particle&
    printDetails( gamma );      // do the same with the Photon
    const Particle& particle_ref = e;  // Put an explicit particle
                                       // reference to Electron
    printDetails( particle_ref );      // And print it
}
```

Summary of Lecture

- I have discussed
  - Basic Constructs (mostly C, a little C++)
    - Basic Types, Loops, Conditionals, Functions, Pointers, Arrays and References
  - Classes and Objects
  - Inheritance
    - Protection and Virtual Functions
  - Interfaces and Default Implementations
    - Virtual and Pure Virtual Functions.

Next Time

- Namespaces
  - demistifying 'using namespace std;'
- Exceptions
- Separating Declarations and Definitions
  - modularising the code by using multiple files
- Standard Template Library taster:
  - vectors, maps and iterators
- Templates ... if we have time

Books

- If you'd like to read more there are some books out there
  - C++ the Core Language 
    - Gregory Satir, Doug Brown, O'Reilly Nutshell series
    - good introduction to basic concepts
  - The C++ Programming Language (3rd Edition)
    - Bjarne Stroustrup
    - The definitive guide to C++ by its inventor 
  - Free books online at
    - http://www.freeprogrammingresources.com/cppbooks.html