

PY410 / 505
Computational Physics 1

Salvatore Rappoccio

C++: Pointers

- The dread cry rang through the night...
NO! POINTERS! NOOOOOO!!!!!!



C++: Pointers

Memory Address	Value
0x1000	0x04a45
0x1001	0x9ab38
0x1010	0x0000
0x1011	0x1003

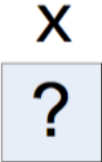
- Actually I've already taught you the concept, just not the syntax
- Pointer is just a variable that holds the **memory address**
- Syntax: "&" operator gives the address:


```
int x = 123;
int y = 456;
int z = 789;
std::cout << "Address of x= " << &x << ", value of x = " << x << std::endl;
std::cout << "Address of y= " << &y << ", value of y = " << y << std::endl;
std::cout << "Address of z= " << &z << ", value of z = " << z << std::endl;
return 0;
```

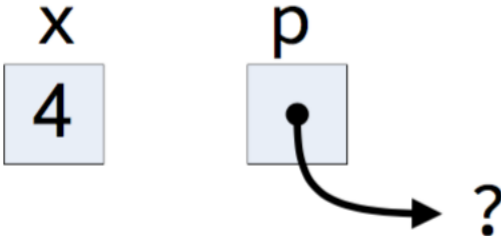
```
Address of x= 0x7fff59364aa8, value of x = 123
Address of y= 0x7fff59364aa4, value of y = 456
Address of z= 0x7fff59364aa0, value of z = 789
```

C++: Pointers

- A pointer variable uses “*”, assign to an address of a variable with “&”:


int x; 

x = 4; 


int *p; 

Read from right to left:
“p is a pointer to an int”

Accessing p here gives
you GARBAGE. It MUST
be initialized!!!

p = &x; 

Assign the pointer to point to a variable by the
ADDRESS operator (&)

*p = 7; 

Access the value of the register POINTED TO by p
by DEREFERENCING (*).

C++: References

- A safer alternative is to have REFERENCES:
 - Pointers that cannot be zero
- A reference variable uses “&”, can treat it like a standard variable
 - But it is not! Be careful! Underlying variable can change!
- Go to “piazza.com”, download “Pointers and References”

C++: Pointers and References

- “ptrs.cc”

```
#include <iostream>

int main(void){

    int x = 10;
    int * px = &x;
    int & rx = x;
    std::cout << "Value of px= " << px << ", dereferenced = " << *px << std::endl;
    std::cout << "Value of rx= " << rx << std::endl;
    *px = 7;
    std::cout << "x = " << x << std::endl;
    rx = 9;
    std::cout << "x = " << x << std::endl;
    return 0;
}
```

- output:

```
Value of px= 0x7fff525a3a98, dereferenced = 10
```

```
Value of rx= 10
```

```
x = 7
```

```
x = 9
```

C++: const with pointers

- You have “const” now with two objects, the variable and the pointer
- There are therefore four possibilities:
- Pointer to int
 - (value can change, pointer can change)
 - `int * p`
- Pointer to const int
 - (value cannot change, pointer can change)
 - `int const * p`
- Const pointer to int
 - (value can change, pointer cannot change)
 - `int * const p`
- Const pointer to const int:
 - (value cannot change, pointer cannot change)
 - `const int * const`

C++: ptrs/refs in functions

- Can use pointers and refs as arguments to functions!
- “ptrs_and_funcs.cc”

```
#include <iostream>
```

```
void increment1( int p){ ++p; std::cout << "p = " << p << std::endl;}  
void increment2( int &p){ ++p; std::cout << "p = " << p << std::endl;}  
void increment3( int *p){ ++(*p); std::cout << "*p = " << *p << std::endl;}
```

```
int main(void){
```

```
    int x = 3;  
    int & rx = x;  
    int * px = &x;  
    std::cout << "0: x = " << x << std::endl;  
    increment1(x);  
    std::cout << "1: x = " << x << std::endl;  
    increment2(rx);  
    std::cout << "2: x = " << x << std::endl;  
    increment3(px);  
    std::cout << "3: x = " << x << std::endl;
```

```
0: x = 3  
p = 4  
1: x = 3  
p = 4  
2: x = 4  
*p = 5  
3: x = 5
```

```
    return 0;
```

```
}
```


C++: ptrs/refs in functions

- Pass by value:

- COPIES value into a temporary variable called “x”

```
void func( int x);
```

- Use when you don't want to modify value, and cheap to copy

- Pass by reference:

- Pass REFERENCE to variable, temporarily called “x”

```
void func( int &x);
```

- Use when you want to modify value, and expensive to copy, ptr=0 disallowed

- Pass by pointer

- Pass POINTER to variable, pointer is called “x”

```
void func( int *x);
```

- Use when you want to modify value, and expensive to copy, ptr=0 allowed

Also usable with **CONST**

C++: Why are pointers hard?

- Dereferencing uninitialized pointers gives you a segmentation fault
 - That's the best case scenario
 - Worst case scenario: It works accidentally, and you accidentally give your credit card information to that Nigerian Prince who keeps emailing you
- Memory management!

C++: Memory Management

- You have access to several pieces of memory:
 - Code: where your code lives
 - Data: static and global variables
 - Stack: static memory
 - local variables and function parameters known at compile time
 - Heap: dynamic memory
 - anything not known at compile time
 - The heap HAS to be accessed via pointer
 - Improperly handling this is a pain
 - The others can be accessed via value or reference

C++: Memory Management

- Allocate on the heap with “new”
- Remove from the heap with “delete”:

```
int i = 123;
int * p = new int( 456 );

std::cout << "i = " << i << std::endl;
std::cout << "*p= " << *p << std::endl;

(*p) += 10;
std::cout << "*p= " << *p << std::endl;

delete p;
```

Allocate an integer off the heap, assign its address to “p”

Do stuff with that heap variable

Remove from the heap

C++: Memory Management

- Every “new” has to come with a “delete”
 - Otherwise you get a **memory leak**
 - Adds memory that does not get cleaned up, eventually your program crashes the computer
- Can be non-obvious
 - What if a function creates a “new” variable and returns it?
 - Still in scope.
 - Stays on the heap.
 - This is called a “factory”

C++: Memory Management

- In modern C++, use “std::auto_ptr” or “std::shared_ptr”
- These will automatically delete the object when the last reference to it goes out of scope
 - I.E. you don't have to worry about the delete operation
 - Also access like a standard pointer:

```
std::auto_ptr<int> pa ( new int(789) );  
std::cout << "*pa=" << *pa << std::endl;
```

Can use the template argument to use ANY type
(more on templates later!)

C++: Arrays and Vectors

- What if you want a group of objects together?

- Arrays (off the stack)

- Intrinsic to C++
- Static at compile time
- Syntax:

```
int array[5] = {0,1,2,3,4};
```

5 elements

Initialized to values here

NOTE: C++ arrays need EITHER a size, OR an initialization, but do not need both if you don't want.

- Vectors (off the heap)

- Part of the Standard Template Library
- Not known at compile time
- Syntax:

```
std::vector<int> vec;
```

- Then use “push_back” to add variables
- (can also “push_front”, etc...)
- more on this later
- In C++0X and C++11: can initialize like an array (see above)
 - When compiling with g++: add “-std=c++0x”

C++: Arrays and Vectors

- Can access individual elements with “[]”:

```
array[1] = 1;
```

- Arrays are just a sequential list of variables
 - Knows nothing about itself.
 - Can only LEGALLY access elements LESS THAN the size of the array!
 - Totally fine with illegal behavior, and will give you garbage
- Vectors are a CLASS, so does know something about itself
 - More on classes later
 - Can therefore:
 - check the size:

```
n = vec.size();
```
 - access elements only if they exist with the “at” method (more later)

```
int list[3];  
list[0] = 5;  
list[1] = -3;  
list[2] = 12;
```

list

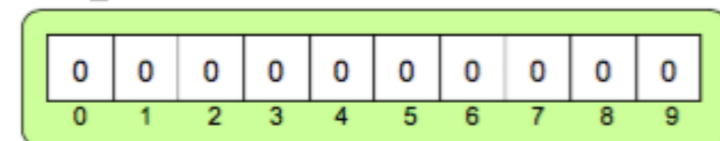
5	-3	12
0	1	2

```
vector<int> vec_a;  
vector<int> vec_b(10);  
vector<int> vec_c(10, 8);  
vector<int> vec_d{ 10, 20, 30, 40 };
```

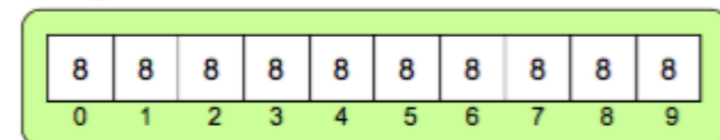
vec_a



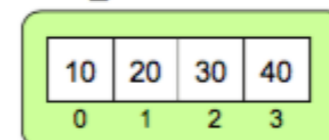
vec_b



vec_c



vec_d



C++: Arrays and Vectors

- N objects in a CONTIGUOUS row of memory:
- For arrays, these are static and from the stack (*)
- For vectors, these are dynamic and from the heap

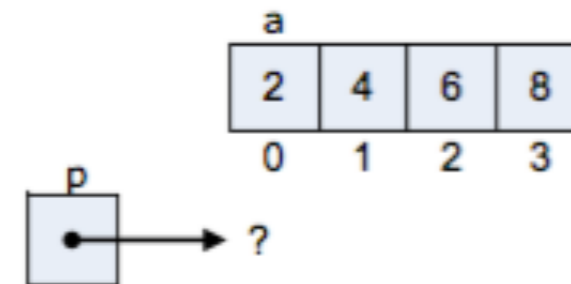
```
array[0] = 0, address = 0x7fff51618990  
array[1] = 1, address = 0x7fff51618994  
array[2] = 2, address = 0x7fff51618998  
array[3] = 3, address = 0x7fff5161899c  
array[4] = 4, address = 0x7fff516189a0
```

- (*)
Technically you can still get arrays off the heap also and do your own dynamic memory allocation. Don't do that. Just use `std::vector`.

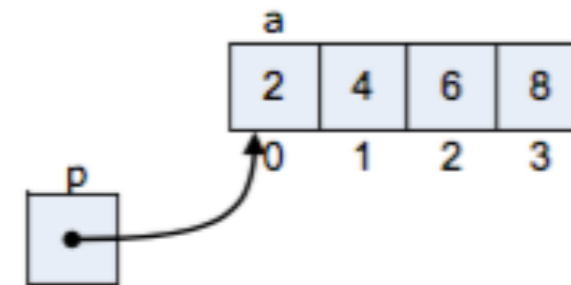
C++: Arrays and Vectors

- Since arrays are just a list of variables, what is the relation between POINTERS and ARRAYS?
- The syntax “a[3]” means:
 - Go to the position 3 variables after the first one
 - But you could just use also use pointers for that!

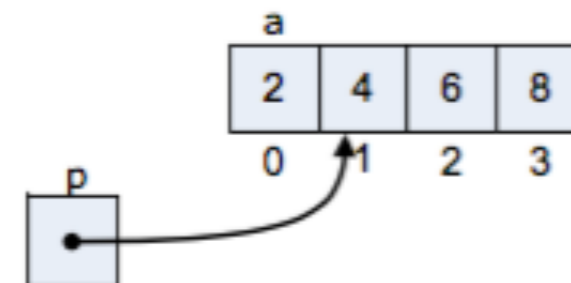
```
int a[] = { 2, 4, 6, 8 }, *p
```



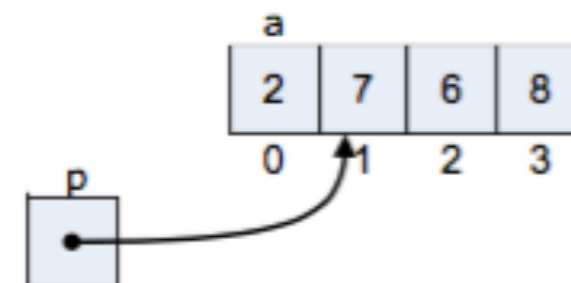
```
p = a;
```



```
p++;
```



```
*p = 7;
```



C++: Arrays and Vectors

- Copying arrays:
 - C++: I have no idea what you're talking about. Do it yourself.

```
int array[5] = {0,1,2,3,4};  
int array2[5];  
for ( unsigned int i = 0; i < 5; ++i ) {  
    array2[i] = array[i];  
}
```

- Copying vectors:
 - C++: Oh! Yeah, sure, no problem!

```
std::vector<int> vec3( vec);
```

C++: Arrays and Vectors

- Multi-dimensional arrays and vectors look like:

```
int M[3][4];
```

3 x 4

- Literally: M is an array of “arrays of size 4”

- Alternatively can use a vector of vectors:

```
std::vector< std::vector<int> > N( 3, std::vector<int>(4) );
```

C++: Arrays and Vectors

- `std::vector` also introduces ITERATORS
- Act like pointers, but are classes (hence smarter)

```
for( std::vector<int>::const_iterator i = vec.begin(); i != vec.end(); ++i ) {  
    std::cout << "i = " << *i << std::endl;  
}
```

- In C++0x and later, can also loop over each item like:

```
for ( int i : vec ) {  
    std::cout << "i = " << i << std::endl;  
}
```

- Why the complication?
 - Faster and safer.

C++: Arrays and Vectors

- Special case of arrays: arrays of “char”
- Similar case as arrays and vectors, `char a[10]` is a fixed-width array (length 10) that can be printed to form characters.
- Then “`std::string`” is similar in spirit to “`std::vector`”
- Moral: use `std::string` when possible.

C++: Command Line Arguments

- Another nice use of arrays: COMMAND LINE ARGUMENTS
- You're already familiar with them (like, "cp old.txt new.txt")
- How to use?

```
int main(int argc, char * argv[] ){
```

- Literally:
 - argc = number of command line arguments
 - I.E. size of array "argv"
 - argv = array of char arrays, each with a string.

C++: Command Line Arguments

- Example: Syntax “commandline.cc”:

```
int main(int argc, char * argv[] ){  
    for ( unsigned int i = 0; i < argc; ++i ) {  
        std::cout << "Argument " << i << " is " << argv[i] << std::endl;  
    }  
}
```

- If our executable is “a.out”, we type on the command line and get:

```
> ./a.out this is how we do it  
Argument 0 is ./a.out  
Argument 1 is this  
Argument 2 is is  
Argument 3 is how  
Argument 4 is we  
Argument 5 is do  
Argument 6 is it
```

- notice: the first argument is the NAME of the executable!

C++: File I/O

- Files in C++ can be opened and closed, in read or write mode.
- The interface to read and write is the same as “std::cout” and “std::cin”.
 - “std::ofstream” : output formatted stream
 - “std::ifstream” : input formatted stream
- Their “parents” (more later) are :
 - “std::ostream” : output stream
 - “std::istream” : input stream

See Chapter 13 of “progcpp.pdf” Textbook for details

C++: File I/O

- Example : copy double from one file to another: “fileio.cc”

```
#include <fstream>
#include <iostream>

int main(void){

    std::ifstream  in("inputfile.txt");
    std::ofstream  out("myfile.txt");
    double d;
    in >> d;
    out << d;
    out.close();

    return 0;
}
```

C++: File I/O

- Within a function, you can use “ostream” and “istream” (“fileio_infuncs.cc”):

```
void input ( std::istream & in ) {
    std::string line;
    std::getline( in, line, ',' );
    std::string firstname = line;
    std::getline( in, line, ',' );
    std::string lastname = line;
    std::getline( in, line );
    int score = std::atof( line.c_str() );

    std::cout << "First name is " << firstname << std::endl;
    std::cout << "Last name is " << lastname << std::endl;
    std::cout << "Score is " << score << std::endl;
}
```

Will need this snippet
for your Homework!