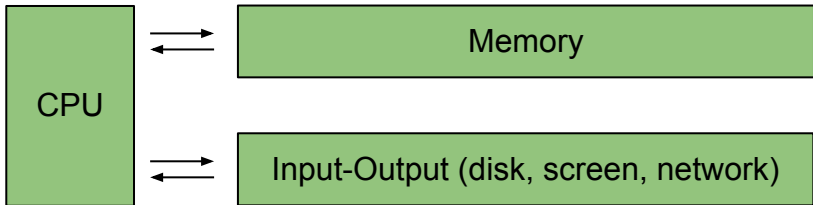


CME 211: Lecture 19

December 4, 2011

- pointers & references
- 3 big ideas
- what's next

Model computer architecture



It is convenient to think of computers as having 3 main blocks: the CPU, memory, and input-output (IO). IO consists of everything that is not the CPU or memory: disk, screen, network, usb, keyboard, etc. The memory stores both instructions for the CPU (code) and data (variables). This diagram is a simplification of the Von Neumann model.

Computer memory



Computer memory can be thought of as a 1D array of bits. Today, most computers address bytes (8 bits). This means that the smallest unit of memory controllable by the programmer is a byte. For example, a C++ `bool` is a byte even though it only uses a single bit, thus 7 bits are wasted. Programmers typically use hex notation for memory addresses. In C and C++ hex numbers have "0x" as a prefix. For performance, access data sequentially. Reads from distant memory locations can be expensive.

Pointers

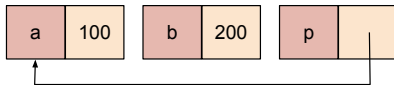
- A pointer is a memory address
- Pointers are typed so that the compiler knows the size of and what to do with the memory at the address
- Some languages don't use pointers: Java, Fortran 77, Python, Matlab
- C uses pointers to support arrays, strings, passing large amounts of data to functions, memory management, etc
- C++ uses pointers because it was designed to extend C
- Pointers are difficult for novice programmers to understand
- Experienced programmers make pointer and memory mistakes all the time

Pointer diagram I

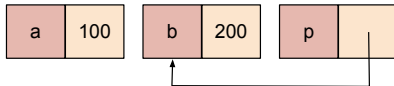
Code

```
double a = 100;  
double b = 200;  
p = &a;
```

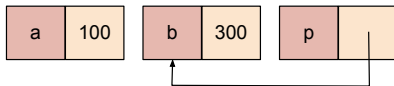
Memory diagram



```
p = &b;
```



```
*p = 300;
```



Pointer diagram II

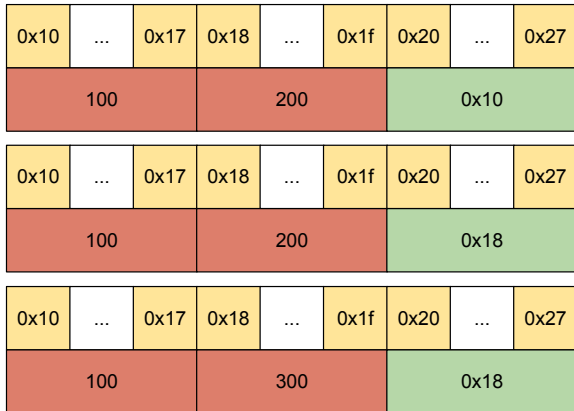
Code

```
double a = 100;  
double b = 200;  
p = &a;
```

```
p = &b;
```

```
*p = 300;
```

Memory layout



Pointers: example

```
1 double a = 100.35;
2 cout << "the_memory_address_of_a:" << &a << endl;
3
4 double b[] = {1, 2};
5 double *p1 = &b[0];
6 double *p2 = &b[1];
7 cout << "address_of_b[0]:_" << p1 << endl;
8 cout << "address_of_b[1]:_" << p2 << endl;
9 cout << "pointer_arithmetic:_" << p2 - p1 << endl;
```

the memory address of a:0x7fff974ab0d8
address of b[0]: 0x7fff974ab0b0
address of b[1]: 0x7fff974ab0b8
example of pointer arithmetic: 1

In C++ we can do a lot without pointers

We all know that pointers are difficult to program correctly and pointer errors are hard to debug. In C++ it is generally better to use the features that replace pointers:

library types C++ vectors, strings, and other library types allow programmers to avoid pointers

iterators objects for accessing elements from a collection of data, like a vector

references used to pass data to a function without copying

Still need pointers when using `new` for memory allocation.

Iterators look like pointers

```
1 vector<int> v;  
2 int N = 10;  
3  
4 for (int i=0; i < N; ++i)  
5     v.push_back(i);  
6  
7 int sum = 0;  
8 for (vector<int>::iterator iter = v.begin();  
9     iter != v.end(); ++iter)  
10     sum += *iter;  
11  
12 cout << "sum:_" << sum << endl;
```

sum: 45

References

In C++, references are generally used as function arguments.

```
1 void ref_swap(int &a, int &b) {  
2     int t = a;  
3     a = b;  
4     b = t;  
5 }
```

```
1 void pnt_swap(int *a, int *b) {  
2     int t = *a;  
3     *a = *b;  
4     *b = t;  
5 }
```

With references, the function can access data with variables names. This is easier than dereferencing. In `pnt_swap` we could change the address of `a` or `b`, which may result in an error. References don't allow this sort of change and are considered safer.

Three Big Ideas

- Abstraction
- Modularization & Encapsulation
- Interfaces

Abstraction

Abstraction is the process of hiding or containing a set of details into a higher-level concept. A good abstraction makes it easier and more efficient for humans to think about and process ideas.

Computers have many layers of abstraction:

- 1 physics, transistors
- 2 logic gates: AND, OR, NAND, XOR...
- 3 logic units, adders, cpus, memory
- 4 cpu instructions
- 5 assembly language
- 6 compiled languages

Modularization & Encapsulation

It is easier to work with systems that are built from well defined and self-contained modules. Modules should be encapsulated, their inner details should be hidden from other modules. Designing such systems is not always easy or quick. However, it generally pays off in the end. These principles should be applied at all levels.

The fundamental tool is a **function**, which should encapsulate a procedure in a meaningful way. The user of the function should generally not need to know the implementation details (ie, sort).

Structs allow the aggregation of data into complex types, which can represent higher level concepts.

Classes facilitate the combination of data and procedures.

Interfaces

Interfaces are the medium through which modules talk to each other. Programmers and users also access things through interfaces. A basic example of an interface is the function argument list and output.

A good interface

- is well defined and documented.
- is easy to use. The amount of required set up code should be minimized.
- provides good defaults if there are many options.

Public vs. Private in Class Types

```
1 class simple {
2 public:
3     int a;
4     void method_1();
5 private: // accessible by class members
6     int b;
7     void method_2();
8 };
9
10 void simple::method_1() {
11     b = 1;
12     method_2();
13 }
14
15 void simple::method_2() {
16     a = 2;
17 }
```

In non class member code:

```
1 simple x;
2 // ok:
3 x.a = 10;
4 x.method_1();
5 // compile time errors:
6 //x.b = 10;
7 //x.method_2();
```

When defining class types, the public and private keywords allow the class designer to specify which member variables and functions are accessible to the user of the function. This allows the class designer to hide implementation details and protect important variables from the user. Any public member of class should be considered as part of the interface.

Common mistake: unexpected integer division

```
1 int x1 = 10;  
2 int x2 = 11;  
3 double z1 = x1/x2;  
4 double z2 = ((double)x1) / x2;  
5 cout << "z1:_" << z1 << endl;  
6 cout << "z2:_" << z2 << endl;
```

z1: 0

z2: 0.909091

Toolset

OS unix, linux, ubuntu

Editor emacs (vim, eclipse, xcode)

Languages python, matlab, C, C++, fortran

Libraries LAPACK, UMFPACK, ARPACK, fftw...

Build system make, cmake, scons, gnu autotools

Tools gdb, gprof, valgrind, git

Other regular expressions, unix tools

References

C/C++

- <http://en.cppreference.com/w/cpp>
- S. B. Lippman, et al. C++ Primer. ([link](#))
- K. N. King. C Programming, A Modern Approach. ([link](#))

Fortran

- <http://fortranwiki.org/>
- <http://fortranwiki.org/fortran/show/Books>

Python

- <http://docs.python.org/tutorial/index.html>
- <http://www.scipy.org/>
- <http://numpy.scipy.org/>

CME 212: MWF 11-12:15

Goal 1: learn computer architecture to write faster code

Goal 2: use tools to write better code faster

- We will cover features of computer architecture and tools to help improve performance
- We'll talk about code design and structure
- We'll study floating point arithmetic in some more detail
- We'll look at tools that aid development: gprof, valgrind, gdb
- Also: LaTeX, regular expressions
- Possibly: building interfaces between Python/Matlab and C/C++/Fortran

For discussion: Linked List

```
1 class my_list {
2 private:
3     struct my_link {
4         int curr_val;
5         my_link *next_link;
6     };
7     my_link *start_link, *last_link;
8     //last_link makes add_element more efficient
9 public:
10    my_list() : start_link(NULL), last_link(NULL) {}
11    ~my_list(); //declaration of destructor
12    void add_element(const int val); //adds an element to the end
13    int get_element(const int n) const;
14    int operator[](const int n) const {return get_element(n);};
15
16    void bubblesort();
17
18    //don't worry about this
19    friend ostream& operator<<(ostream& stream, const my_list &list);
20};
```