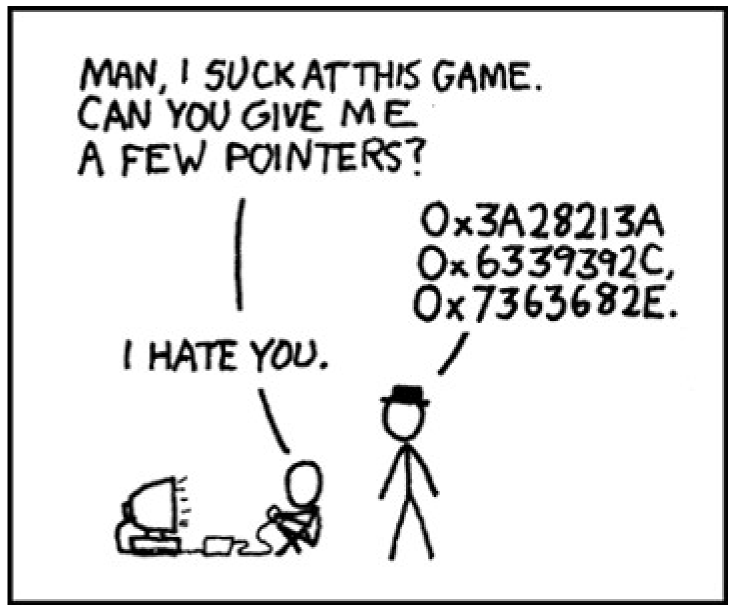
C++ Pointers (Part I)

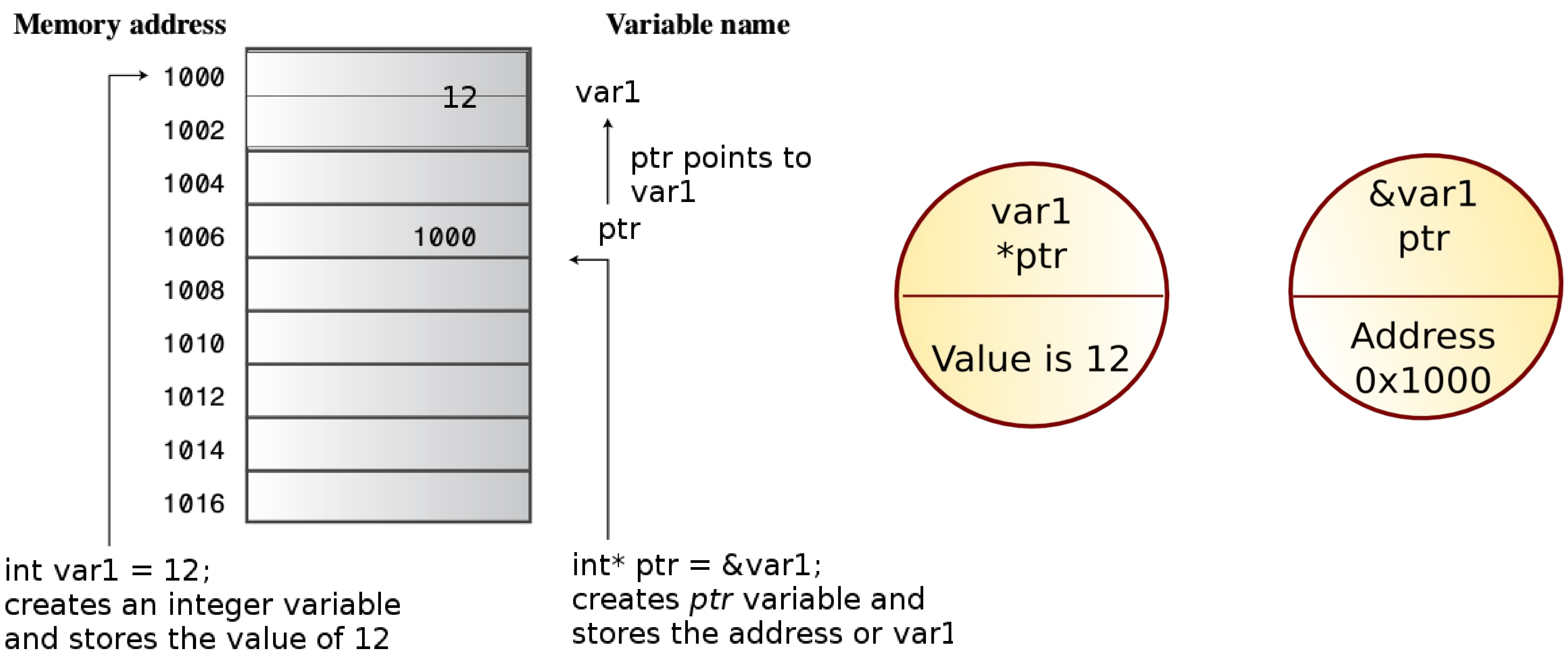


Krishna Kumar

Every computer, at the unreachable memory address 0x-1, stores a secret. I found it, and it is that all humans ar-- SEGMENTATION FAULT
XKCD (https://xkcd.com/138/)

Pointer

- Pointer values are memory addresses
  - A pointer **p** can hold the address of a memory location
  - A pointer points to an object of a given type
    - E.g. a ***int**** points to an ***int***, not to a ***string***



Pointer to

```
char* cptr;                // Pointer to a char
int* iptr;                 // Pointer to an int
float* fptr;               // Pointer to a float
MyClass* myclasspt;        //  Pointer to a user-defined class MyClass
int* ap[15];               // array of 15 pointers to ints
int (*fp)(char*);          // pointer to function taking a char*
                           // argument; returns an int
int* f(char*);             // function taking a char* argument; returns a
                           // pointer to int
```

Pointer Danger

int* ptr; // create a pointer-to-int

****ptr = 556;*** // place a value in never-never land

- ***Pointer Golden Rule:*** Always initialize a pointer to a definite an appropriate address before you apply the dereferencing operator (*) to it

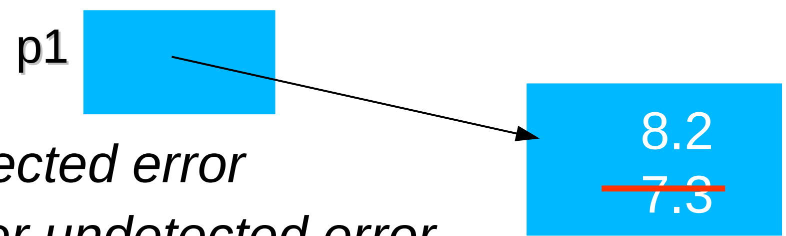
Capsule Summary

```
int v;            //defines variable v of type int
int* p = &v;      //defines p as a pointer to int
                  //assigns address of variable v to
                  // pointer p
v = 3;            //assigns 3 to v
*p = 3;           //also assigns 3 to v
```

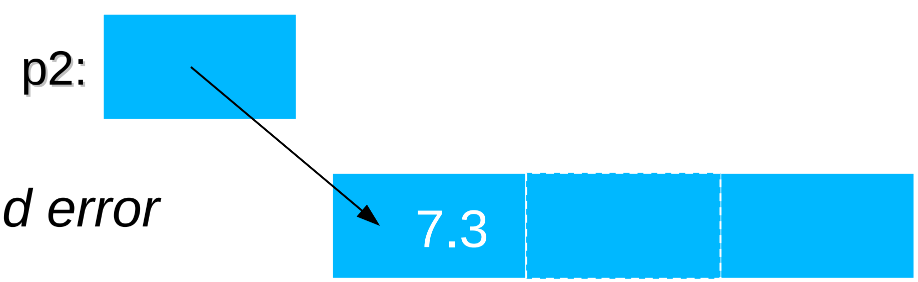
Access

- A pointer does **not** know the number of elements that it's pointing to (only the address of the first element)

```
double* p1 = new double;
*p1 = 7.3;            // ok
p1[0] = 8.2;          // ok
p1[17] = 9.4;         // ouch! Undetected error
p1[-4] = 2.4;         // ouch! Another undetected error
```




```
double* p2 = new double[100];
*p2 = 7.3;            // ok
p2[17] = 9.4;         // ok
p2[-4] = 2.4;         // ouch! Undetected error
```

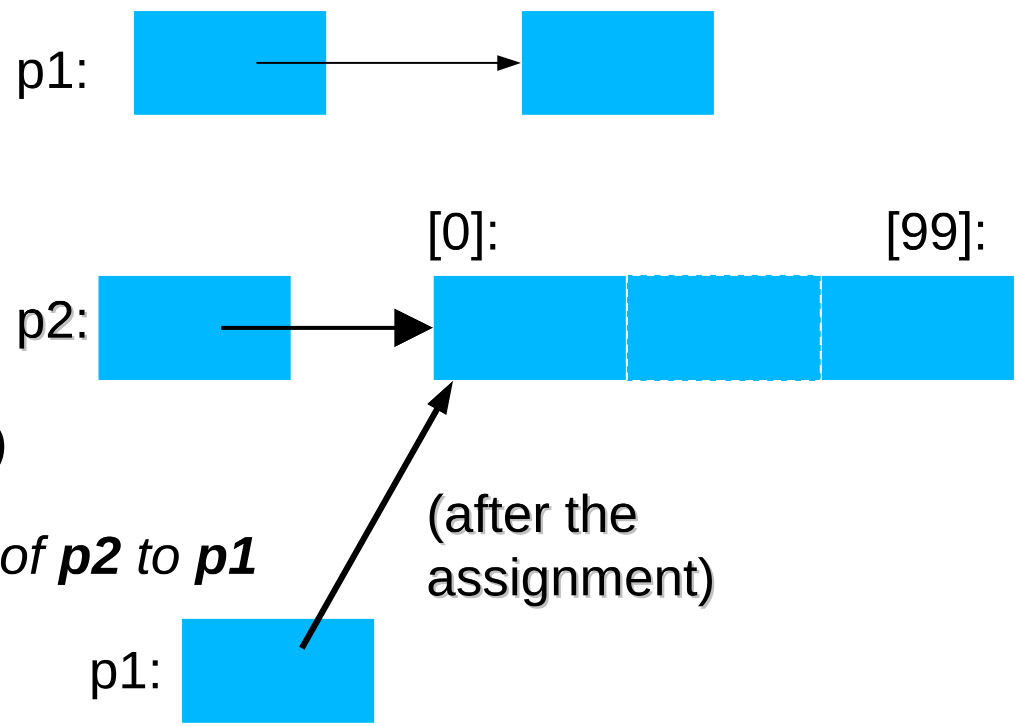


Access

- A pointer does ***not*** know the number of elements that it's pointing to

double* p1 = new double;

double* p2 = new double[100];



p1[17] = 9.4; *// error (obviously)*

p1 = p2; *// assign the value of **p2** to **p1***

p1[17] = 9.4; *// now ok: **p1** now points to the array of 100 **doubles***

Access

- A pointer ***does*** know the type of the object that it's pointing to

```
int* pi1 = new int(7);
int* pi2 = pi1;        // ok: pi2 points to the same object as pi1
double* pd = pi1;      // error: can't assign an int* to a double*
char* pc = pi1;        // error: can't assign an int* to a char*
```

  - There are no implicit conversions between a pointer to one value type to a pointer to another value type
  - However, there are implicit conversions between value types:

```
*pc = 8;      // ok: we can assign an int to a char
*pc = *pi1;   // ok: we can assign an int to a char

int * pi;
pi = 0xB8000000;   // type mismatch (You know that's an address compiler doesn't)
pi = (int* ) 0xB8000000;     // types now match ( int 2 Bytes and addr may be 4Bytes)
```

Computer Memory

- **Stack:**
  - Stores local data, return addresses, used for parameter passing
- **Heap:**
  - You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data
- **Use local variables (stack) when you can. Use dynamic allocation (heap) when you have to.**

Stack Memory

- Variables when out of scope will **automatically be deallocated.**
- **Faster to allocate** in comparison to variables on the heap.
- Can have a **stack overflow** when too much of the stack is used.
- Data created on the stack **can be used without pointers.**
- Use stack when data is not too big.

Heap Memory

- Variables on the heap must be **destroyed manually** and **never fall out of scope**.
- The data is freed with **delete, delete[] or free**
- **Slower to allocate** in comparison to variables on the stack.
- Data can be accessed by pointers
- Can have allocation failures if too big of a buffer is requested to be allocated.
- You would use the heap if you don't know exactly how much data you will need at runtime or if you need to allocate a lot of data.
- **Responsible for memory leaks**

The free store
(sometimes called "the heap")

- You request memory "to be allocated" "on the free store" by the **new** operator
  - The **new** operator returns a pointer to the allocated memory
  - A pointer is the address of the first byte of the memory
  - For example
    - **int* p = new int;** *// allocate one uninitialized **int***
      *// **int*** means “pointer to **int**”*
    - **int* q = new int[7];** *// allocate seven uninitialized **int**s*
      *// “an array of 7 **int**s”*
    - **double* pd = new double[n];** *// allocate **n** uninitialized **double**s*
  - A pointer points to an object of its specified type
  - A pointer does ***not*** know how many elements it points to

Why use free store or heap?

- To allocate objects that have to outlive the function that creates them:
  - For example

```
double* make(int n)     // allocate n ints
{
    return new double[n];
}
```

  - Another example: vector's constructor
- Huge data (usually greater than >1MB)

Pointer Danger
(when using new)

- Individual elements

```
int* p1 = new int;          // get (allocate) a new uninitialized int
int* p2 = new int(5);       // get a new int initialized to 5

int x = *p2;                // get/read the value pointed to by p2
                            // (or "get the contents of what p2 points to")
                            // in this case, the integer 5
int y = *p1;                // undefined: y gets an undefined value; don't do that
```

Stack and Heap Memory

```
int foo() {
    int* ptr; //<--nothing allocated yet (excluding the pointer itself, which is
                // allocated here on the stack).
   bool b = true; // Allocated on the stack.
   if(b)  {
    //Create 500 bytes on the stack
    int buffer[500];
    //Create 500 bytes on the heap
    ptr = new int[500];
   }  //<-- buffer is deallocated here, pBuffer is not
return 0;
}  //<--- oops there's a memory leak, I should have called
   // delete[]  ptr;
```

A problem: memory leak

```
double* calc(int result_size, int max) // function returns pointer to a  double
{
    double* p = new double[max];        // allocate another max doubles
                                        // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    return result;
}

double* r = calc(200,100);              // oops! We "forgot" to give the memory
                                        // allocated for p back to the free store
```

- Lack of de-allocation (usually called "memory leaks") can be a serious problem in real-world programs
- A program that must run for a long time can't afford any memory leaks

A problem: memory leak

```
double* calc(int result_size, int max)
{
    int* p = new double[max];   // allocate another max doubles
                                // i.e., get max doubles from the free store
    double* result = new double[result_size];
    // … use p to calculate results to be put in result …
    delete[ ] p;                // de-allocate (free) that array
                                // i.e., give the array back to the free store
    return result;
}

double* r = calc(200,100);
// use r
delete[ ] r;                    // easy to forget
```

Memory leaks

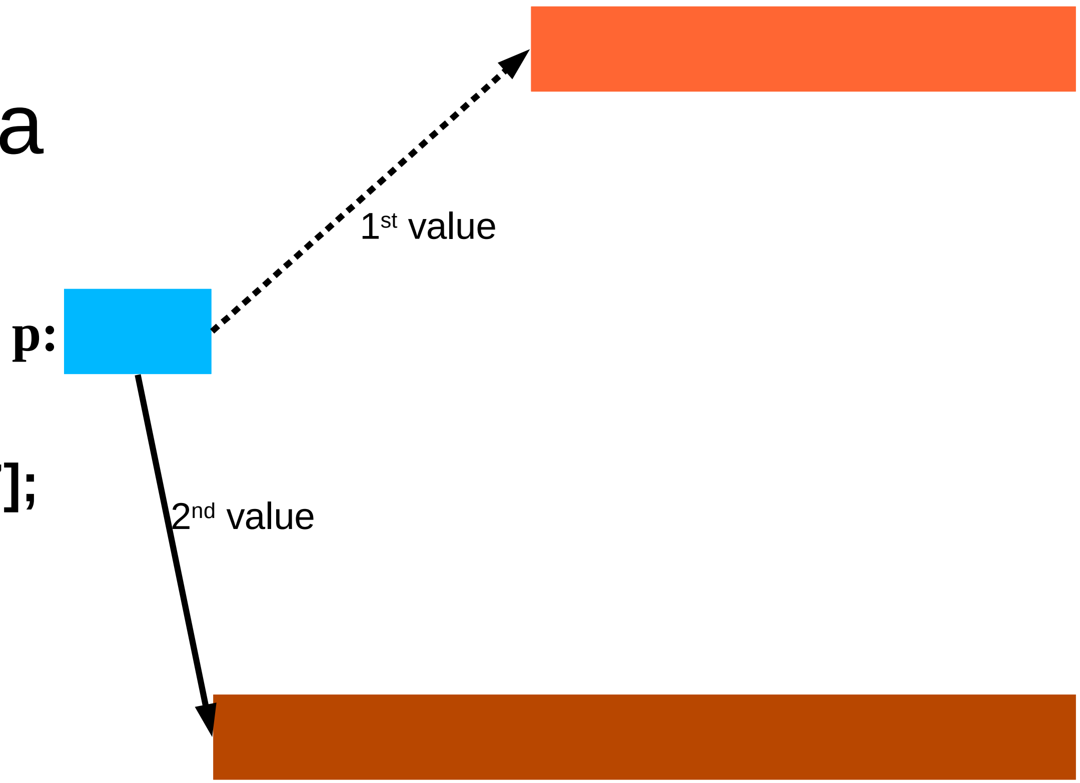
- A program that needs to run "forever" can't afford any memory leaks
  - An operating system is an example of a program that “runs forever”
- If a function leaks 8 bytes every time it is called, how many days can it run before it has leaked/lost a megabyte?
  - Trick question: not enough data to answer, but about 130,000 calls
- All memory is returned to the system at the end of the program
  - If you run using an operating system (Windows, Unix, whatever)
- Program that runs to completion with predictable memory usage may leak without causing problems
  - *i.e.,* memory leaks aren’t “good/bad” but they can be a major problem in specific circumstances

Memory leaks

- Another way to get a memory leak

```
void f()
{
    double* p = new double[27];
    // …
    p = new double[42];
    // …
    delete[] p;
}
```

// 1st array (of 27 doubles) leaked



Memory leaks

- How do we systematically and simply avoid memory leaks?
  - don't mess directly with **new** and **delete**
    - Use **vector**, etc.
  - Or use a garbage collector
    - A garbage collector is a program the keeps track of all of your allocations and returns unused free-store allocated memory to the free store (not covered in this course; see http://www.research.att.com/~bs/C++.html)
    - Unfortunately, even a garbage collector doesn't prevent all leaks
  - Use smart_ptr (auto, unique_ptr, shared_ptr)

Vector class: memory leak

```
class vector;

void f(int x)
{
   vector v(x);     // define a vector
   // (which allocates x doubles on the free store)
   // … use v …

   // give the memory allocated by v back to the free store
   // but how? (vector's elem data member is private)
}
```

Vector (destructor)

```
// a very simplified vector of doubles:
class vector {
    int sz;                    // the size
    double* elem;              // a pointer to the elements
public:
    vector(int s)                          // constructor: allocates/acquires memory
        :sz(s), elem(new double[s]) { }
    ~vector()                  // destructor: de-allocates/releases memory
        { delete[ ] elem; }
    // …
};
```

- **N**ote: this is an example of a general and important technique:
  - acquire resources in a constructor
  - release them in the destructor
- Examples of resources: memory, files, locks, threads, sockets

A problem: memory leak

```
void f(int x)
{
    int* p = new int[x];   // allocate x ints
    vector v(x);           // define a vector (which allocates another x ints)
    // … use p and v …
    delete[ ] p;   // deallocate the array pointed to by p
    // the memory allocated by v is implicitly deleted here by vector's destructor
}
```

- The **delete** now looks verbose and ugly
  - How do we avoid forgetting to **delete[ ] p**?
  - Experience shows that we often forget
- Prefer **delete**s in destructors

Free store summary

- Allocate using **new**
  - New allocates an object on the free store, sometimes initializes it, and returns a pointer to it
    - **int* pi = new int;** *// default initialization (none for **int**)*
    - **char* pc = new char('a');** *// explicit initialization*
    - **double* pd = new double[10];** *// allocation of (uninitialized) array*
  - New throws a **bad_alloc** exception if it can't allocate
- Deallocate using **delete** and **delete[ ]**
  - **delete** and **delete[ ]** return the memory of an object allocated by **new** to the free store so that the free store can use it for new allocations
    - **delete pi;** *// deallocate an individual object*
    - **delete pc;** *// deallocate an individual object*
    - **delete[ ] pd;** *// deallocate an array*
  - Delete of a zero-valued pointer ("the null pointer") does nothing
    - **char* p = 0;**
    - **delete p;** *// harmless*

References

- Bjarne Stroustrup C++
- C++ Primer Plus
- Object Oriented Programming in C++
- http://people.ds.cam.ac.uk/nmm1/C++/index.html
- http://www.thecodingforums.com/threads/stack-or-heap-for-c-object.689680/
- https://stackoverflow.com/questions/79923/what-and-where-are-the-stack-and-heap#79936