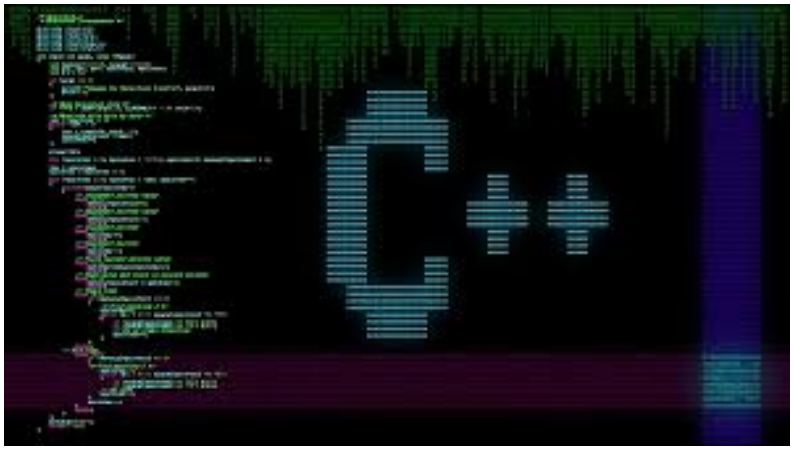
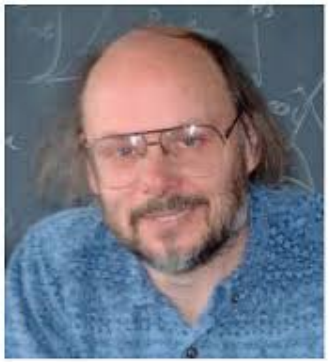
C++

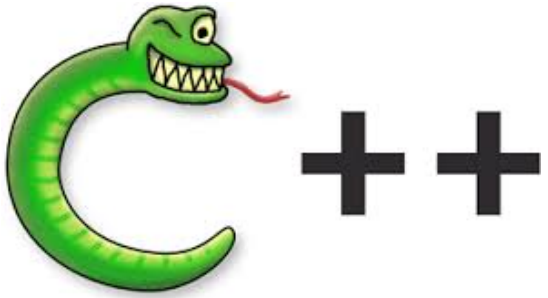
CS 1
Introduction to Computer Programming

Lecture 23: November 30, 2015

C++, part 2







Last time

- Essentials of C++
  - Compiling C++ programs
  - Comments
  - Functions and the `main()` function
  - Data types
  - Variables
  - Operators
  - Expressions and statements
  - Input/output
  - Conditionals: `if`, `else`, `else if`
  - `while` and `for` loops

Today

- The "hard" stuff!
  - Addresses and pointers
  - Arrays
  - Pointer arithmetic
  - The stack and the heap (free store)
  - Dynamic memory allocation: `new` and `delete`
  - Pitfalls

Addresses

- All data in C++ programs has an associated location in memory (as in all languages)
- C++ is unusual in that you can access that location as a language value
- An *address* is a location in memory where data can be stored
  - a variable, an array, an object, *etc.*
- Address of variable `x` is written `&x`
- Addresses are like integers, but they obey different rules

Pointers

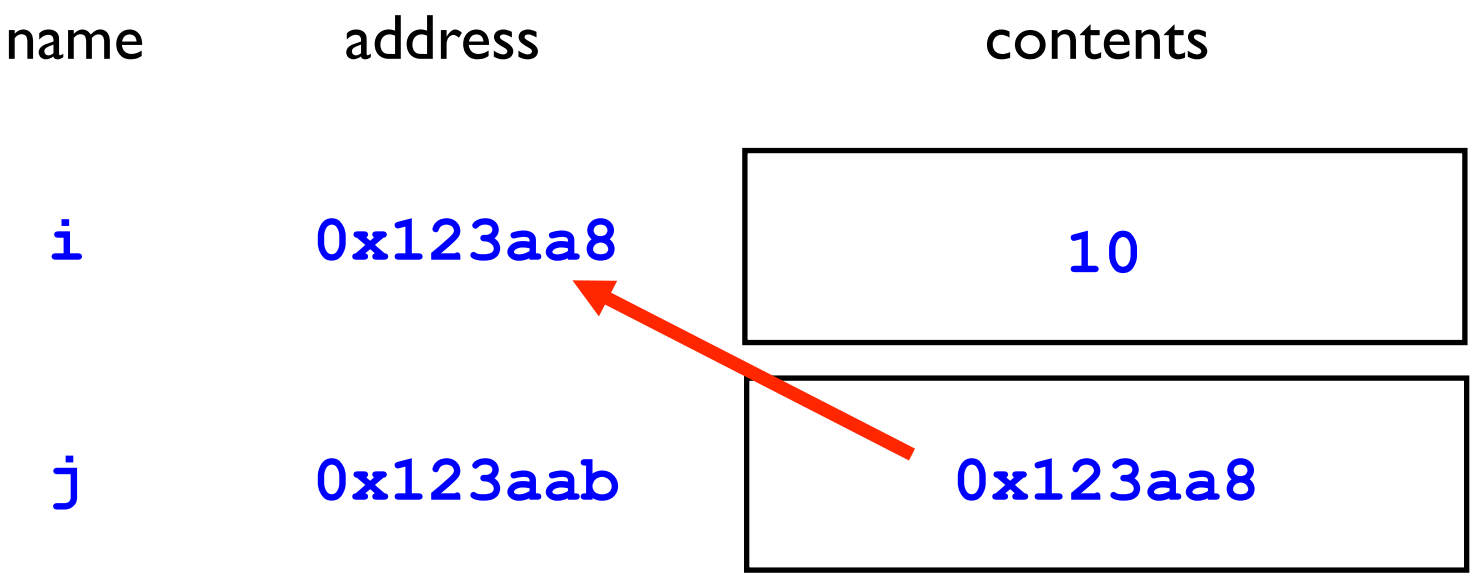
- Pointers cause a great deal of confusion for new C++ programmers
- However, the concept of a pointer is very simple:

A pointer is a variable which contains an address

Pointers

- Why are pointers useful/interesting/ relevant?
  - They allow *indirect access to data* (through the address of the data)
  - They can be used to *optimize* code in ways that would be impossible without them
  - They are intimately connected to the C++ notion of an *array*
  - They are used when *allocating memory dynamically* (with the **`new`** and **`delete`** operators)

Pointers in action




```
int  i = 10;
int *j = &i;    // j "points" to i
```

Pointers in action

```
int  i = 10;

int *j = &i;

cout << "i = " << i << "\n";

cout << "j = " << j << "\n";

cout << "j points to: " << *j << "\n";
```

Pointers in action

- Results:

```
i = 10

j = 0x123aa8

j points to: 10
```

Pointers in action

- `&i` is the address of variable `i`
- `*j` is the contents of the address stored in pointer variable `j`
  - *i.e.* the value of what `j` "points to"
- `*` operator dereferences the pointer `j`

Pointers and the * operator

- The many meanings of the * operator:
  - Multiplication (duh)

```
a =  b * c;
```

  - *Declaring* a pointer variable

```
int *a;
```

  - *Dereferencing* a pointer

```
cout << *a << "\n";
```

Pointer pitfalls (1)

- Declaring multiple pointer variables:

```
int *a, *b;  // a, b are ptrs to int
```

- If you do this:

```
int *a, b;   // b is just an int
```

- Then only the first variable will be a pointer
- Rule: *every* pointer variable in declaration must be preceded by a *****

Pointer pitfalls (2)

- Note that

```
int *j = &i;
```

- really means

```
int *j; // j is a pointer to int
j = &i; // assign i's addr to j
```

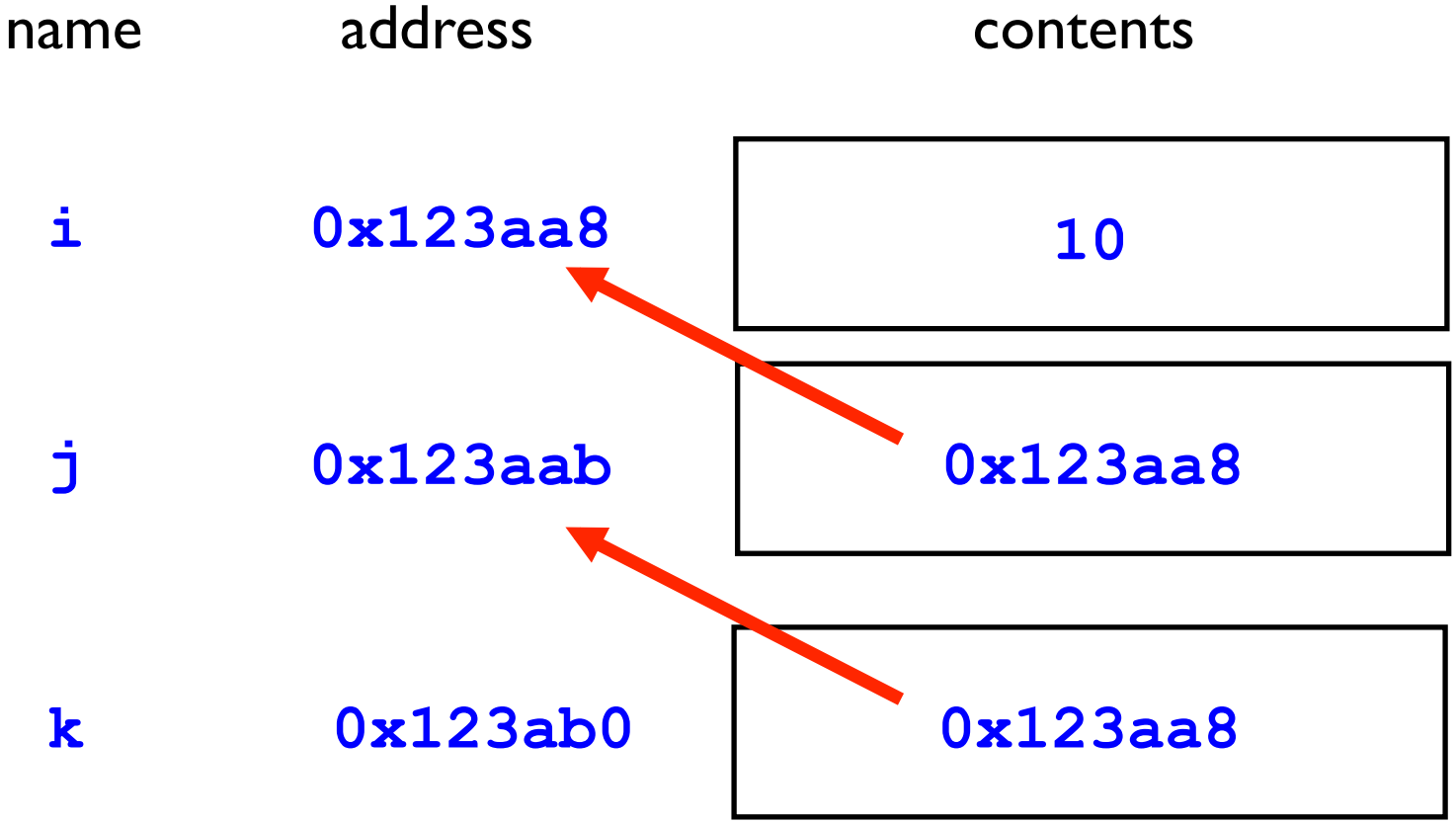
- Don't confuse this `*j` with a dereference!

Pointers to pointers

- A harder problem:

```
int   i = 10;
int  *j = &i;
int **k = &j;
cout << &i << "\t" << i << "\n";
cout << &j << "\t" << j << "\t"
     << *j << "\n";
cout << &k << "\t" << k << "\t"
     << *k << "\t" << **k << "\n";
```

Pointers to pointers



Pointers to pointers

- We can have pointers, and pointers to pointers, *etc.* as far as we want to go
  - We can't have "addresses of addresses" though…

```
int   i = 10;  // regular int
int  *j = &i;  // ptr to int
int **k = &j;  // ptr to ptr to int
```

- A pointer to `int` requires one dereference (`*j`) to get to the value of the `int`
- A pointer to a pointer to an `int` requires two dereferences (`**k`) to get to the value of the `int`

Assigning to pointers

```
int i = 10;
int *j = &i;
int *m;
// Assign to what j points to:
*j = 20;  // Now i is 20.
// Assign j to m:
m = j;    // Now m points to i too.
// Assign to what j points to:
*j = *m + i;  // Now i is 40.
```

Assigning to pointers

- When pointer variable is on left-hand side of an assignment statement, what happens depends on whether it's dereferenced or not
- No dereference: assign the value on RHS (an address) to the pointer variable on the LHS

```
j = m;
```

Assigning to pointers

- When pointer variable is on left-hand side of an assignment statement, what happens depends on whether it's dereferenced or not
- Dereference: assign value on RHS into location corresponding to where pointer points to

```
*j = *m + 10;
```

Assigning to pointers

- When pointer variable is declared and assigned to on the same line:

```
int *j = m;
```

- it means:

```
int *j;  // declare j
j = m;   // assign to j
```

- *i.e.* assign the RHS to the pointer variable on the LHS
- RHS must be an address, or it's a type error!

Mnemonics: fetch/store

- When you use the `*` (dereference) operator in an expression, you *fetch* the contents at that address

```
cout << "j's contents: " << *j << "\n";
```

- When you use the `*` (dereference) operator on the left-hand side of the `=` sign in an assignment statement, you *store* into that address

```
*j = 42;   // store 42 into address
```

Arrays in C++

- Arrays in C++ are conceptually similar to lists in Python
- They store a sequence of values which can be accessed using the square bracket notation: `arr[0]`, `arr[1]` *etc.*
- However, internally they are very different!

Arrays in C++

- Python arrays are "smart"
- They store their length (`len`), they have associated methods (`append` *etc.*) and operators (`+` for concatenation *etc.*)
- C++ arrays are "dumb"
  - really, really, *really* dumb!
  - *no* methods, *no* operators, just indexing
  - they don't know how big they are
  - technically, they don't even exist!

Arrays in C++

- There is a "smart" array type in C++ called a *vector*
- For practical use, vectors are far superior to arrays, but…
- … vectors are built using arrays, so you still need to understand arrays!

Arrays in C++

- Another difference: C++ arrays can only contain values of *one* type per array
- We speak of an "array of `int`s", an "array of `float`s", an "array of `char`s" *etc.*
- Can't store anything but an `int` into an array of `int`s!

Declaring arrays

- We have to declare arrays before using them
- Syntax looks like this:

```
int arr[5];  // declare array of 5 ints
```

- Or we can initialize an array when created:

```
int arr[5] = {1, 2, 3, 4, 5};
int arr[] = {1, 2, 3, 4, 5}; // also ok
```

- If we don't initialize the contents of an array when we declare it, it contains…
- … garbage! (Whatever values were there before)

Declaring arrays

- You can also declare 2-D arrays (and higher dimensional arrays):

```
int arr2[2][3];   // values are garbage
int arr2[2][3] =
   { { 1, 2, 3 }, { 4, 5, 6 } };
```

- That's all we'll say about 2-D arrays!

Using arrays

- Arrays are used as you might expect:

```
int arr[] = {1, 2, 3, 4, 5];
cout << "arr[0] = " << arr[0] << "\n";
arr[0] = 42;
```

- Nothing interesting to see here, folks… run along…

Arrays and pointers

- Actually, there is no such thing as an "array" in running C++ code!
- In other words, an "array" is not a "data structure" per se
- It is just a *chunk of memory*, along with some *pointer* that contains the address of that chunk of memory!
- The memory has to be in consecutive locations (addresses) in the computer

Arrays and pointers

- C++ does *not* keep track of the size of any chunk of memory that corresponds to what we would call an array
- Therefore, you can do horrible things like this:

```
int arr[] = {1, 2, 3, 4, 5};
cout << arr[1000000] << "\n"
```

- … and your program will probably crash (no exception raised)

Arrays and pointers

- So what is the deal with arrays and pointers?
- Consider this innocent-looking line:

```
int arr[10];
```

- What actually happens when this executes is:

```
int *arr;
arr = <chunk of memory>;
```

- where `<chunk of memory>` is big enough to hold 10 `int`s

Arrays and pointers

- In other words, even though we didn't say anything about pointers, creating an array created a pointer to the location at the beginning of the array!
- When we refer to the name of the array by itself (`arr`) we are referring to that pointer
- Similarly, `&(arr[0])` is equal to `arr`
- (OK to write this as just `&arr[0]`)

Pointer arithmetic

- You can also use pointers "manually" with arrays
- The way to do this involves *pointer arithmetic*
- Idea: you can add/subtract integers to pointers (addresses) to get other pointers (addresses)
- However, this arithmetic doesn't obey normal arithmetic rules!

Pointer arithmetic

- When you add `1` to a pointer variable (address) that contains the address of an array element, you get the *address of the next item in the array*, but the numeric value of the address will usually change by more than `1`
- (In fact, the numeric value of the address will increment by the size of the item in bytes)

Pointer arithmetic

- Consider:

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

- Now:
  - `arr == &arr[0]`
  - `*arr == arr[0] == 1`
  - `arr + 1 = &arr[1]`
  - `*(arr + 1) = arr[1] == 2`
  - *etc.*
- The numeric difference between `arr` and `arr + 1` depends on the size of an `int` in bytes

Pointer arithmetic

- So when the C++ compiler sees `arr[n]`, it converts it automatically to `*(arr + n)`
- (At least, for simple types with no user-defined operator overloading of the `[]` syntax)
- (C++ is very complicated, so it's hard to make blanket statements!)

Pointer arithmetic

- This equivalence between `arr[n]` and `*(arr + n)` also holds when assigning to array items:

```
arr[n] = 10;
*(arr + n) = 10;  // same thing!
```

- Idea: compute the location to be changed `(arr + n)`, then use dereferencing with assignment to change it

Array arguments

- This equivalence between pointers and arrays also holds when passing arrays as arguments to functions:

```
int sum_array(int arr[], int len) {
    result = 0;
    for (int i = 0; i < len; i++) {
        result += arr[i];
    }
    return result;
}
```

- *N.B.* We have to pass a **`len`** argument!

Array arguments

- We could also write this as:

```
int sum_array(int *arr, int len) {
    result = 0;
    for (int i = 0; i < len; i++) {
        result += arr[i];
    }
    return result;
}
```

- This means the exact same thing!
- Maybe better (less ambiguous, more obvious why you need a `len` argument)

Pointer arithmetic

- Since arrays are just chunks of memory and pointers to those chunks, you can see that it's easy for array bounds violations to happen
- Nothing checks that an array access is within those bounds!
- Go too far out of bounds, and the operating system will shut down your program for using another program's memory
  - "Segmentation violation"!

The stack and the heap

- Another critical aspect of C++ programming is the distinction between the *stack* and the *heap*
- They are two distinct kinds of memory locations, with very different behaviors
- Technically, the "heap" in C++ is more correctly called the "free store" for very boring reasons

The stack and the heap

- Chunks of memory (*e.g.* for arrays) can be allocated on either the stack or the heap, but the syntax and the behavior of this memory is different!
- We need to understand this difference, because it's a key cause of confusion for new C++ programmers

The stack

- We talked about the runtime stack when we were talking about exception handling previously
- Virtually all programming languages have a runtime stack of some sort
  - Python, Java, C, C++, etc.
- A key feature of the stack is that it stores *stack frames* which are collections of local variable values for a single call to a function

The stack

- Python manages all memory allocations and deallocations automatically, so we don't have to worry about it
- C++ doesn't, so we do!

The stack

- Local variables of "simple" types (`int`, `float`, `double`, `char`) are allocated on the stack:

```
int k = 10;  // allocated on the stack
```

- Once the function where this line occurs in returns, the memory where these variables are located is *automatically deallocated*
- This automatic deallocation is the key feature of the stack from a memory standpoint

The stack

- The arrays we've been working with are also allocated on the stack:

```
int arr[10];  // allocated on the stack
```

- This means that you can't just return this array as the return value of a function!
- Actually, you *can*, but you don't want to!
- If you use it, you will be using memory that has already been deleted and possibly re-used
- Very nasty kind of bug!

The stack

- Note that you *can* return values of simple types from functions with no problems
  - Simple types are copied from one function to another
  - *N.B.* Pointers are considered simple types too
- But "compound" types like arrays are not copied when returned from a function
  - Too expensive to do that!
  - So there has to be a different way to deal with this

The heap (free store)

- Problem: We often want to create an array in a function and return it so other functions can use it
- In Python, this doesn't require any special handling
- In C++, it does!
- You need to allocate memory in a different part of memory, called the "heap" or "free store"
  - "free store" is the technically correct term

The heap (free store)

- The important things about the heap:
  - allocating heap memory is manual
  - heap memory is *never* deallocated unless you specifically ask it to be
- Therefore, it's OK to return heap-allocated memory from a function
- C++ introduces two new keywords to deal with this: **`new`** and **`delete`**
- **`new`** allocates memory on the heap
- **`delete`** deallocates memory from the heap

`new` and `delete`

- Let's start by allocating a single integer on the heap:

```
int *h = new int;
```

- Or we can give it an initial value:

```
int *h = new int(42);
```

- This memory will not be deallocated when the function it's part of returns!
- We have to deallocate it manually using `delete`:

```
delete h;
```

`new` and `delete`

- **`new`** always returns a pointer to the given datatype:

```
new int(42)  // returns int *
```

- **`delete`** should *only* be called on a pointer that was returned from **`new`**:

```
int *h = new int(42);
// ... use *h ...
delete h;
```

`new` and `delete`

- Inside this function, we need to use `*h` to access the value of the integer:

```
int *h = new int(42);
cout << "*h = " << *h << "\n";
delete h;
```

- This will print:

```
*h = 42
```

`new` and `delete`

- **`new`** and **`delete`** also work with arrays:

```
int *arr = new int[5]; // allocate array of 5 ints
// Now arr contains garbage.
// Initialize the elements of arr,
// then do something with arr.
// Finally, delete the memory.
delete[] arr;
```

- For this, need to call **`delete[]`** instead of **`delete`**
- This is one of the commonest uses of **`new`**

`new` and `delete`

- A different approach:

```
int *arr = new int[5]();
// Now arr elements all have their default
// values (0 in this case).
// Do something with arr.
// Finally, delete the memory.
delete[] arr;
```

- The parentheses cause the "default constructor" to be called for each `int` in the array; for `int` this means that all elements are `0` to start with

`new` and `delete`

- This is not legal:

```
int *arr = new int[5] {1, 2, 3, 4, 5}; // nope
```

- If you want this, you need to initialize the array elements manually
- Reasons are beyond the scope of this lecture!
  - Which means: I don't really understand it myself ☺

delete

- **`delete`**/**`delete[]`** take a single argument, which is a pointer to memory allocated using **`new`**
- **`delete`** doesn't "change" the pointer
  - the numerical value of the pointer is the same before and after calling **`delete`**
- Instead, it informs the system that the memory pointed by the pointer can be re-used for later calls to **`new`**
- After calling **`delete`**, you had better not use that memory again!

Separate functions

- Usually, we call `new` in one function to create some kind of data, and we call `delete` or `delete[]` in another function to get rid of the data once we've finished using it
- Let's see an example…

Separate functions

```
int *range(int start, int end) {
    int len = end - start;  // no error checking
    int *result = new int[len];
    for (int i = 0; i < len; i++) {
         arr[i] = start + i;
    }
    return result;
}
```

- This function returns a new array, allocated on the heap
  - Actually, it returns a pointer to the first element of the new array

Separate functions

```
void range_test() {
    int *result = range(0, 10);
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += result[i];
    }
    cout << "sum = " << sum << "\n";
    delete[] result;
}
```

- This function calls **`range`** to get a new array
- It computes and prints the array sum
- It **`delete`**s the array once it's done

Separate functions

```
void range_test() {
    int *result = range(0, 10);
    int sum = 0;
    for (int i = 0; i < 10; i++) {
        sum += result[i];
    }
    cout << "sum = " << sum << "\n";
    delete[] result;
}
```

- *N.B.* **`void`** means that the function doesn't return anything

Pitfalls

- Many, many things can go wrong when using pointers, **`new`**, **`delete`** and **`delete[]`**
- Some of these will give rise to incorrect values
- Some will crash your program!
- Some will cause your program to use more memory as it goes on, until you eventually run out of memory
- C++ is challenging!

Pitfalls

- Pointers that don't point to valid objects sometimes called "dangling pointers"
- Consider this:

```
int *arr;
cout << arr[0] << "\n";
```

- Error?
- No value at the location `arr[0]`!

Pitfalls

- Consider this:

```
int *f() {
    int arr[] = {1, 2, 3};
    return arr;
}
// later...
int *some_array = f();
cout << some_array[0] << "\n";
```

- Error?
- Accessing deallocated (stack) memory!

Pitfalls

- Consider this:

```
int *arr = new int[3];
arr[0] = 1; arr[1] = 2; arr[2] = 3;
delete[] arr;
cout << arr[0] << "\n";
```

- Error?
- Accessing deleted memory!

Pitfalls

- Consider this:

```
void f() {
    int *arr = new int[3];
    arr[0] = 1; arr[1] = 2; arr[2] = 3;
    cout << arr[0] << "\n";
}
```

- Error?
- **`delete[]`** is never called in **`f()`**!
- This is a "memory leak"

Memory leaks

- Memory leaks occur when memory from the heap (free store) is allocated with `new` but `delete` or `delete[]` is never called on that memory
- Over time, the program may use up more and more memory
- If the program runs long enough, the program may run out of memory and a crash will happen in an otherwise innocuous place

Memory leaks

- Memory leaks are *extremely* hard to prevent
- C++ class system makes it somewhat easier by allowing *destructors* (methods that deallocate all memory allocated when creating an object)
  - Kind of like the `__del__` method in Python, but more important

Memory leaks

- Problem: pointers can be copied anywhere without copying memory they point to
- It's hard to know exactly when to `delete` memory (and when it isn't safe)
- Many programmers just give up and don't bother deleting memory at all
  - This kind of sloppy programming invariably leads to memory leaks
- Deleting memory too aggressively can lead to deleting memory still in use (even worse!)

Memory leaks

- Special tools exist to debug memory leaks *e.g.* Valgrind (`http://valgrind.org`)
- Take CS11 advanced C++ track to learn more about this

Pitfalls

- Consider this:

```
int *arr = new int[3];
arr[0] = 1; arr[1] = 2; arr[2] = 3;
cout << arr[0] << "\n";
delete arr;
```

- Error?
- Calling **`delete`** instead of **`delete[]`**!

`delete` and `delete[]`

- Rule: When you allocate memory with **`new`** that *isn't* an array, you deallocate it with **`delete`**, not **`delete[]`**
- Rule: When you allocate memory with **`new`** that *is* an array (some authors call this **`new[]`**), you deallocate it with **`delete[]`**, not **`delete`**

Pitfalls

- Consider this:

```
int *arr = new int[3];
arr[0] = 1; arr[1] = 2; arr[2] = 3;
int *arr2 = &arr[1];
cout << arr2[0] << "\n";
delete[] arr2;
```

- Error?
- Calling **`delete[]`** on a pointer not returned from **`new`**!

Pitfalls

- You should only call **`delete`** or **`delete[]`** on the *exact* pointer value returned from **`new`**
- If you change that pointer value, don't call **`delete`**/**`delete[]`** on it
- Doing so may make your program crash!
- For arrays: either you **`delete[]`** the *entire* array, or *none* of it
- Can't go to *e.g.* the middle of the array and **`delete[]`** the second half!

Summary

- C++ requires that you manage memory very carefully
- Must understand how addresses and pointers work
- Must choose whether to allocate/deallocate memory on the stack or the heap (free store)
- Must match
  - `new` with `delete`
  - `new arr[]` with `delete[] arr`

Summary

- Lots of subtleties I didn't have time to go into!
- Mostly relating to how **`new`**/**`delete`**/ **`delete[]`** work with classes and objects
- Hopefully CS2 will tell you what you need to know about these topics ☺

Next time

- Advanced topics, part 1
  - recursion
  - first-class functions
  - `lambda` expressions
  - higher-order functions
    - `map`, `filter`, `reduce`