Dynamic memory with C++

- No need to use `malloc` or `free` anymore
  - Use `new` and `delete` – to manage the "free store"
- No need to specify the size either
  - Just `int *ip = new int;` // compiler knows sizeof(int)
    - `*ip = 7;` // must dereference pointer to use
    - … `delete ip;` // "If you allocate it, you release it"
  - Or `int &ir = *new int;` // "Name that space!"
    - `ir = 4;` // no need for pointer notation – `ir` is a reference
    - … `delete &ir;` // but releasing still requires a pointer

Arrays on the free store

- `int *ptr = new int[5];` // array of 5 `int`s
  - Size of array can be an expression
  - No way to initialize array elements!
    - Must iterate, and assign to each one individually
- `delete [] ptr;` // use `[]` to delete array
- Array of C strings requires array of `char *`
  - So array pointer is `char **`
  - Allocate space for each `char` array – release in reverse order
  - Multidimensional arrays are similarly handled
- See …`/demo06/newarray.cpp`, `cstrings.cpp` and study text section titled "Do You Understand?" on pp. 69-70

C++ structures: `struct`, `class`

- Either one can have data *and functions*

```
struct Square {  // or class Square
    int side;
    double area() const  // supports constant objects
    {  return side * side; }
};
```

- Only 2 differences – both in default accessibility
  - e.g., both `side` and `area()` are `public` above – both would be `private` if `class` instead of `struct`
  - Other small difference is in default inheritance access

Declaring and defining classes

- `class Foo;` // just a declaration
  - Sometimes all you need – usually in header files
  - e.g., need `ostream &` – `#include <iosfwd>`
- `class Foo { … };` // a definition
  - Note: some or all implementation likely elsewhere
- Usual definition style is most to least accessible

```
public:  // the public interface is listed first
    int getValue();
protected:  // for subclasses (like Java) – omit if not used
private:  // most data should be here – listed last
    int value;
```

Implementing classes

- Usually in a separate file – `foo.cpp`, not `foo.h`
  - So `#include "foo.h"`
- Identify class with scope resolution operator

```
int Foo::getValue() { return value; }
```

- Implementation can include other stuff too
  - Use helper functions, data, constants, even classes
  - No worries about name conflicts in other files
- Usually one implementation file per public class

Using class instances – objects

- Declare to create on stack or global space
  - `Foo foo1, foo2;` // created two `Foo`s
- Or use `new` to create on free store
  - `Foo *fooPtr = new Foo;` // one more `Foo`
- Contact object directly with `.` operator
  - `foo1.getValue();`
- Or through a pointer with `->` operator
  - `fooPtr->getValue();`
- See `point-circle` example in …`/demo06/`