Chapter - 27
From C to C++

Upgrading

There is a lot of C code out there. It is 95% compatible with C++.

That's why we're studying the other 5%.

K&R Style Functions

```
                {
        // Body of the function


            char *name;
        int function;
                {
        // Body of the function

int funct(...);     // Default prototype for class C functions


            i = do_it();
        i = do_it(1, 2, 3);
    i = do_it("Test", 'a');
```

enum, struct, union and class

C++

```
struct sample {
    int i,j;    // Data for the sample
};
sample sample_var;  // Last sample seen
```

C

```
struct sample sample_var;  // Legal in C
sample sample_var;         // Illegal in C
```

malloc and free

C's version of **new** is `malloc`:

```
foo_var = (struct foo *)malloc(sizeof(struct foo));
```

and `calloc` (`calloc` zeros the data)

```
foo_var = (struct foo*)calloc(3, sizeof(foo));
/* C++ uses     foo_var = new foo[3] */
```

C++ `malloc` trap:

```
class foo {...};
foo_var = (struct foo *)malloc(sizeof(struct foo));
// Don't code like this
```

WARNING: This creates the class without calling the constructor.

```
free((char *)foo_var);
foo_var = NULL;
```

WARNING: This destroys the class without calling the destructor.

Turning Structures into Classes

Structure reading and writing

```
a_struct struct_var;

// Perform a raw read to read in the structure
read_size = read(fd, (char *)&struct_var, sizeof(struct_var));

// Perform a raw write to send the data to a file
write_size = write(fd, (char *)&struct_var, sizeof(struct_var));
```

Class reading and writing (NOT)

```
class sample {
public:


sample(void) : sample_size(100) { } // Set up class
virtual void get_sample(); // Routine to get a sample
};

sample a_sample;
// ...
read_size = read(fd, (char *)&a_sample,
```

Zeroing Structures and Classes

Clearing a structure

```
struct a_struct { ... }
a_struct struct_var;
// ...
memset(&struct_var,'\0',sizeof(struct_var));
```

Clearing a class -- NOT!

```
class a_class { ... }
a_class class_var;
// ...
memset(&class_var, '\0', sizeof(class_var));
```

setjmp

```
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Where:

`env` is the place where setjmp saves the current environment for later use by longjmp.

Returns

0 Normal call

Non-zero

Non-zero return codes are the result of a `longjmp` call.

longjmp

```
void longjmp(jmp_buf env,
             int return_code);
```

Where:

`env` is the environment initialized by a previous setjmp call.

`return_code` is the return code that will be returned by the `setjmp` call.

setjmp / longjmp usage

```
#include <setjmp.h>
#include <iostream>

jmp_buf location; // Place to store location data
void subroutine() {
    class list a_list;
    // Exception found, use
    longjmp(location, 5);

    // This code is never executed
}
int main() {
    if (setjmp(location) == 0) {
        std::cout << "Normal execution\n";
        subroutine();
    } else {
        std::cout << "Exception\n";
    }
```

3) Execption found longjmp called to handle emergency exit

Code here is not executed and this include the destructor for a_list

1) Normal setjmp call (return value = 0)

2) Normal suborutine() call

4) setjmp return value=5 from longjmp

5) Exception handled

Turning C into C++

1. Change K&R style function headers into standard C++ headers.
2. Add prototypes.
3. Change `setjmp/longjmp` calls into **catch/throw** operations.

Following these two steps you have a C+1/2 program. It works, but it's really a C program in C++'s clothing. To convert it to a real C++ program you need to do the following.

4. Change `malloc` into **new**.
5. Change `free` into delete or **delete []** calls.
6. Turn `printf` and `scanf` calls into `std::cout` and `std::cin`.
7. When turning **struct** declarations into **class** variables be careful of `read`, `write` and `memset` functions that use the entire structure or class.