Legacy Constructs

Kate Gregory
www.gregcons.com/kateblog

pluralsight
hardcore developer training

Legacy Code

- **Code never really goes away**
  - Printed in books
  - Released in open source libraries
  - Blogs, articles, and similar internet repositories
- **How you react to it depends on what you want it for**
  - Learning
  - Maintaining
  - Using (eg call a library)
  - Adapting
- **As a beginner:**
  - Using is safe
  - Learning – be particular
  - Maintaining and adapting are not for you

What makes it Legacy?

- **Reasonably modern C++ but pre C++ 11**
  - No lambdas
  - No ranged for
- **Written by someone who didn't like templates**
  - No std::vector
  - No static_cast
- **Written when habits were different**
  - No const
  - Lots of manual memory management
  - Pointers everywhere
- **No Standard Library**
  - No std::string
  - No std::cout
- **C being compiled as C++**
  - No classes

Casting

- **You may see this:**

```
int i = (int) 4.9;
```

- **That means the same as**

```
int i = static_cast<int>( 4.9);
```

- **Other kinds of cast unfortunately also use the same syntax**

```
p = (Employee*) input;
x = (Transaction) t;
```

- **It's very hard to know what kind of cast it is**
- **Don't copy this when you see it – just know what it means**

Macros

- **An old C construct**

```
#define Pi 3.14
```

- **When defining a constant, use a const variable with a type:**

```
const double Pi = 3.14;
```

  - Ideally in a class to avoid name collisions

- **Sometimes used to make "functions"**

```
#define SQR(x) (x*x)
```

- **Nightmare if written wrongly**
  - `SQR(1+1)` will expand to `(1+1*1+1)` which is 3
- **Instead, use a function**
  - Inlining will make it nice and fast

C Style Arrays

- **Declared with a type, and an extent known at compile time**

```
int numbers[4];
```

- **Can be initialized when declared**

```
double morenumbers[] = {1.1,2.2,3.3,4.4,0};
```

  - Extent is deduced by the compiler
- **These arrays do not know how many elements they hold**
  - Developer needs to handle that somehow
- **When you see [] you probably think vector**
  - C style arrays are reasonably similar
  - But they can't grow themselves
  - Can also be accessed with pointer manipulation
  - Code that uses the array name alone is getting a pointer to the start of the array

C Style strings

- **Before `std::string` there were still strings**
  - They were just way more work to use
- **A C-style array of `char` elements**
  - Sometimes called “char* strings”
- **With a special “signal” element at the end**
- **The character with the numeric value zero**
  - `'\0'`
  - **Not `'0'` or `"0"`**
  - Requires an extra space at the end to hold this null terminator
- **Dozens of functions to work with these kinds of strings**
  - strlen
  - strcpy
  - strcat
- **A literal string like `"Hello"` is actually represented like this**
  - But luckily std::string knows how to construct from a char* string
- **When you see char*, or char and [], think “string”**
- **Also when you see functions that start str**

Printf

- **The C way of writing to the screen**
  - Means “print with format”
- **First parameter is a “format string”**
  - Mix of literal text and placeholders

```
printf("%s() called\nBaseaddress: 0x%08x\n",
        symbol_name, base_address);
printf("Returnaddress: 0x08%x\n", return_address);
printf("In File: %s (Line: %d)\n\n",
        file_name, line_number);
```

- **When reading someone else’s code, you can generally ignore the specific meaning of each placeholder (%d, %f, %x etc)**

Typedef

- **Sets up synonyms that appear to be new types, but are just new names for existing ones**

```
typedef int BOOL
```

  - You'll see this in code that predates the bool type in C++
- **Sometimes done to make code more portable**
  - In theory, can just change the typedef and remaining code is unchanged
- **Sometimes done for expressivity**
  - Distinguish between an `int` that represents a size and one that represents a position
- **Sometimes to reduce typing or <> fear**

```
typedef std::vector<int> VectorOfInt;
VectorOfInt numbers;
typedef std::vector<double> VectorOfDouble;
VectorOfDouble morenumbers;
```

Pointers

- **Too much here to cover but punctuation to look for:**
  - * in a declaration means pointer

```
int* pi; // pi is a pointer-to-int
int *pi; // no difference
```

  - & before a variable means "address of"

```
int j = 4;
pi = &j;
```

  - * before a variable name means "content of"

```
*pi = 3; //changes j's value
```

  - Incrementing a pointer makes it point somewhere else
  - Generally only done with arrays
  - `->` after a pointer-to-object is like `(*pointer).`
    - Calling member function
- **Pointer arithmetic is a great source of bugs**
  - Don't copy and reuse such code
  - Understand it and redo using constructs you know (eg string, vector)

Summary

- **Old code has its uses**
  - But when it uses old constructs, it can be hard to read
- **It can also be a great source of bugs**
  - Never copy-and-paste without understanding
  - Try to represent your understanding using modern constructs
- **Much of C++'s reputation for difficulty came from these old constructs**
  - You don't have to use them!
  - There are safer, easier alternatives available
- **Try looking for example code that uses a more modern approach**