Code Editing in Local Style

by Peter E Conn

UNIVERSITY OF CAMBRIDGE

The Problem

- There are over 20 different coding styles for C.
- Various open source projects use different styles.
- Most programmers have their own favourite style.
- This raises the barrier for entry to an open source project and negates some of the benefits of having a defined style in the first place.
- Most of them differ over similar aspects: naming scheme, white space, etc.

The Solution

CELS, a tool which allows interconversion between different coding styles.

- **CELS** allows the user to convert to/from a set of predefined styles, such as FreeBSD, Linux and GNU.
- The user can create a specification for their own, preferred style.
- These features together allow users to contribute to a project in a foreign style, without getting bogged down adjusting to the new style.
- **CELS** is based on the stable libclang API, so doesn't depend on a specific version of LLVM.

Pattern Based Symbol Renaming

CELS can perform a pattern based rename, bringing an entire code collection into line with a specified coding standard.

- Different Naming Schemes can be specified per symbol type.
- Works across multiple files.
- Only renames symbols whose definition is given in the sources - so it ignores symbols defined in external libraries and APIs.
- Allows user defined patterns/naming schemes.
- Keeps track of inconsistent symbol names in the original document - so a conversion to a naming scheme and back is lossless.

Symbol types that can be renamed:

- Variable Names
- Function Names
- Struct/Union Names
- Enum Names
- Enum Constant Names

Built In Naming Schemes:

- lowerCamelCase
- UpperCamelCase
- lower_underscore
- CAPS_UNDERSCORE
- lower

White Space

CELS can deal with the white space aspects of code formatting, including:

- Dealing with indents (better than existing tools - correctly handles macros)
- Correcting inter-symbol spacings, such as "**int *x;**" and "**int* x;**"
- Creating pretty alignment (eg, aligning functions names with different return types).

Optimal Line Breaking

CELS uses a custom-made dynamic-programming algorithm (inspired by TeX) to insert line breaks in the optimal places.

- Each pair of tokens is assigned a line breaking penalty.
- The total penalty over the set of lines is minimized.
- The user can define the scores for pairs of tokens.

Align Parameters, (*, '&&') has minimum penalty:

```
if(GetCondition(parameter1,
                parameter2)
   && TriggerSet(flag1,
                 flag2)
   && safety < threshold){
```

Don't Align Params, ('&&', *) has minimum penalty:

```
if(GetCondition(parameter1,
   parameter2) &&
   TriggerSet(flag1,
   flag2) &&
   safety < threshold){
```

Symbol Renaming

```
int square(int X){
  return X*X;
}

int main(){
  int LEN = 3;
  return square(LEN);
}
```

```
int Square(int x){
  return x*x;
}

int main(){
  int len = 3;
  return Square(len);
}
```

Converts symbols names to different styles, differentiating based on the type of symbol.

Declaration Split/Merge

```
int main(){
    int x = 0, y;
    float i, *j;

    return x;
}
```

```
int main(){
    int x = 0;
    int y;
    float i;
    float *j;

    return x;
}
```

Opening Brace Moving

```
int main(){
    int x = 0;
    return x;
}
```

```
int main()
{
    int x = 0;
    return x;
}
```

Putting It Together

```
#include <stdio.h>

int nthPrime(int n);

int main()
{
    for(int i=1; i<20; i++)
    {
        printf("%d: %d\n", i,
                nthPrime(i));
    }

    return 0;
}

int isPrime(int n)
{
    if(n < 2)
        return 0;

    for(int d = 2;d < n; d++)
    {
        if(n % d == 0)
            return 0;
    }

    return 1;
}

int nthPrime(int n)
{
    int i;

    for(i=0; n; i++)
    {
        if(isPrime(i))
            n--;
    }

    return i-1;
}
```

```
#include <stdio.h>

int is_prime(int n);
int nth_prime(int n);

int main(){
    int i;

    for(i=1; i<20; i++){
        printf("%d: %d\n", i,

                nth_prime(i));
    }

    return 0;
}

int is_prime(int n){
    int d;

    if(n < 2)
        return 0;

    for(d = 2;d < n; d++){
        if(n % d == 0)
            return 0;
    }

    return 1;
}

int nth_prime(int n){
    int i;

    for(i=0; n; i++){
        if(is_prime(i))
            n--;
    }

    return i-1;
}
```

Prototype Generation

```
int main(){
    int x = 0;
    return x;
}

int Square(int x){
    return x*x;
}
```

```
int Square(int x);

int main(){
    int x = 0;
    return x;
}

int Square(int x){
    return x*x;
}
```

Works Across Files

CELS can track renames across an entire code-base.

Declaration Moving

```
int main(){
    int x = 10;
    int t;

    while(x--){
        t=x;
        while(t--)
            printf("-");
        printf("\n");
    }

    return 0;
}
```

```
int main(){
    int x = 10;

    while(x--){
        int t;
        t=x;
        while(t--)
            printf("-");
        printf("\n");
    }

    return 0;
}
```

Moves declarations between the start of the function and the most local scope.

Consistency Checking

CELS is capable of checking a code base to ensure that it meets a certain style, giving a list of errors for every point the style is violated.
This is designed to be integrated with open source projects, so the more mundane aspects of code review can be automated.

Style Inferring

CELS can be run over a code base to infer the coding style used.
Then, by using the Checker, **CELS** can report occasions when the style is not followed, or by using the Converter, **CELS** can correct the violations.

Future Plans

Additional Features:

- Conversion between typedefs and Hungarian (eg. "**int f_verbose;**" <-> "**typedef flag int; flag verbose;**").
- Extension to support Objective-C and C++.

Integration:

- With a VCS, so local code can be converted to project style when committed/pushed.

The source code is available under the FreeBSD license at:
https://bitbucket.org/PEConn/cels .