UCanCode FlowChart/Diagram Component

YOU CAN COD
software components wit

With The Case Of UCanCode.net Release The Power OF Visual C++ !

UCanCode UCanCode.NET VISUALIZATION TOOLKIT HMI CAD GIS FLOWCHART

Download Free Trial
Pricing & Purchase?
E-XD++Visual C++/ MFC Products

- Overview
- Features Tour
- Electronic Form Solution
- Visualization & HMI Solution
- Power system HMI Solution
- CAD Drawing and Printing Solution
- Bar code labeling Solution
- Workflow Solution
- Coal industry HMI Solution
- Instrumentation Gauge Solution
- Report Printing Solution
- Graphical modeling Solution
- GIS mapping solution
- Visio graphics solution
- Industrial control SCADA &HMI Solution
- BPM business process diagramming Solution
- Organization Diagram Solution
- Graphic editor Source Code
- UML drawing editor Source Code
- Map Diagramming

ActiveX COM Products
- Overview
- Download
- Purchase

Technical Support
- General Q & A
- Discussion Board
- Contact Us

Links

UCanCode.NET Find out more...
Get Ready to Unleash the Power of UCanCode .NET

UCanCode Software focuses on general application software development. We provide complete solutior matter you want to develop a simple database workflow application, or an large flow/diagram based syst provide a complete solution for you. Our product had been used by hundreds of top companies around tl

"100% source code provided! Free you from not daring to use components because of unable to maste of components!"

Free VC++ Tool: Convert C++ Code file to HTML

By q123456789.

A utility which **generate** or **converts** your **C++ code to HTML**.

[**Download source and exe files - 69.1 Kb**](#)

Introduction

Cpphtml is a utility to **convert your C++ code to HTML**. If you have a **C++ file**, say *myprogram.cpp*, it on your website, you can run it through *Cpphtml* which will **convert the code to HTML** with all comm preprocessor directives highlighted. *Cpphtml* will send all output to `cout`, so you have to redirect the ou want to create a HTML file:

```
C:\>cpphtml myprogram.cpp >myprogram.htm
```

Cpphtml will convert all tabs to 4 spaces. If you want the tab size to be 8 spaces, you can specify the command line:

```
C:\>cpphtml myprogram.cpp 8 >myprogram.htm
```

The **HTML** code contains a `<style>` element which contains style rules for comments, keywords and pre So, you don't have to do a search-and-replace if you want to change, say, the color of keywords. For all keywords in bold, just change the `.keyword` style rule: `.keyword{color:rgb(0,0,255);font-weight:bold`

I don't claim *Cpphtml* works perfectly. I tested it on the Dinkumware STL files, the source of *Cpphtml*, CPP file. The results are great. *Cpphtml* was compiled with the Borland C++ 5.5 command line compiler:

A walk through the code

```
#include<fstream>
#include<string>
#include<ctype.h>
```

Cpphtml will replace all tabs by 4 spaces if no tab size is specified. Change `_TABSIZE` to 8 if you want th be 8.

```
#define _TABSIZE    4

using namespace std;

int tabsize = _TABSIZE;
```

Token is a class which represents chunks of code. A token can have the type *comment*, *pp (preproces keyword* or *code*. *Code* is everything which is not a *comment*, *pp* or *keyword*. Note that there are no g methods: because `operator>>` and `operator<<` are friends of class token, we don't need any.

```
class token {
public:
    token() : _what(code) {}
protected:
```

→ Purchase

```
    friend ostream& operator<<(ostream&, const token&);
};
```

The function `iskeyword()` returns `true` if string `s` is a **C++** keyword, `false` if not. It's possible you don't r
keywords, e.g. `and`. Those keywords can be used by programmers who don't have access to all ASCII cl
seen code with such keywords though.

⊟ Collapse

```
bool iskeyword(const string& s)
{
    static const char* keywords[] = {
        "and",
        "and_eq",
        "asm",
        "auto",
        "bitand",
        "bitor",
        "bool",
        "break",
        "case",
        "catch",
        "char",
        "class",
        "compl",
        "const",
        "const_cast",
        "continue",
        "default",
        "delete",
        "do",
        "double",
        "dynamic_cast",
        "else",
        "enum",
        "explicit",
        "export",
        "extern",
        "false",
        "float",
        "for",
        "friend",
        "goto",
        "if",
        "inline",
        "int",
        "long",
        "mutable",
        "namespace",
        "new",
        "not",
        "not_eq",
        "operator",
        "or",
        "or_eq",
        "private",
        "protected",
        "public",
        "register",
        "reinterpret_cast",
        "return",
        "short",
        "signed",
        "sizeof",
        "static",
        "static_cast",
        "struct",
        "switch",
        "template",
        "this",
        "throw",
        "true",
        "try",
        "typedef",
        "typeid",
        "virtual",
        "void",
        "volatile",
        "wchar_t",
        "while",
        "xor",
        "xor_eq"
    };

    for (int i = 0; i < sizeof(keywords) / sizeof(char*); i++)
        if (string(keywords[i]) == s)
            return true;

    return false;
}
```

The function `containspp()` returns `true` if string `s` contains a substring which is a preprocessor directive. can contain a string of the form "`#...define`", therefore, we have to find a substring.

```
bool containspp(const string& s)
{
    static const char* pptokens[] = {
        "define",
        "elif",
        "else",
        "endif",
        "error",
        "if",
        "ifdef",
        "ifndef",
        "include",
        "line",
        "pragma",
        "undef"
    };

    for (int i = 0; i < sizeof(pptokens) / sizeof(char*); i++)
        if (s.find(pptokens[i]) != string::npos)
            return true;

    return false;
}
```

`Operator>>` extracts a token from an input stream. It recognizes "`//`" and "`/*...*/`" comments, preproces form "`#...define`", and keywords. String constants are also recognized to avoid keywords to be highlight

⊟ Collapse

```
istream& operator>>(istream& is, token& t)
{
    t._str = "", t._what = token::code;
    int c = is.get();
    switch (c) {
        case '/':
            c = is.get();
            if (c == '*') {
                t._str = "/*";
                t._what = token::comment;
                while (1) {
                    c = is.get();
                    if (c == EOF)
                        return is.unget(), is.clear(), is;
                    if (c == '/') {
                        if (t._str.length() > 2 &&
                            t._str[t._str.length() - 1] == '*') {
                            return t._str += '/', is;
                        }
                    }
                    t._str += (char)c;
                }
            } else if (c == '/') {
                t._str = "//";
                t._what = token::comment;
                c = is.get();
                while (c != '\n' && c != EOF) {
```

```
            t._str += '\n';
        }
        return is;
    }
    t._str = '/';
    return is.unget(), is.clear(), is;
case '#':
    t._str = '#';
    c = is.get();
    while (strchr(" \r\n\t", c)) {
        t._str += (char)c;
        c = is.get();
    }
    if (c == EOF)
        return is.unget(), is.clear(), is;
    while (strchr("abcdefghijklmnopqrstuvwxyz", c)) {
        t._str += (char)c;
        c = is.get();
    }
    is.unget(), is.clear();
    if (containspp(t._str))
        t._what = token::pp;
    return is;
case '\'':
case '"': {
    char q = (char)c;
    t._str = q;
    while (1) {
        c = is.get();
        if (c == EOF)
            return is.unget(), is.clear(), is;
        if (c == q) {
            if (t._str.length() >= 2) {
                if (!(t._str[t._str.length() - 1] == '\\' &&
                    t._str[t._str.length() - 2] != '\\'))
                    return t._str += q, is;
            } else {
                return t._str += q, is;
            }
        }
        t._str += (char)c;
    }
}
case 'a':
case 'b':
case 'c':
case 'd':
case 'e':
case 'f':
case 'g':
case 'i':
case 'l':
case 'm':
case 'n':
case 'o':
case 'p':
case 'r':
case 's':
case 't':
case 'u':
case 'v':
case 'w':
case 'x':
    t._str += (char)c;
    c = is.get();
    while (isalpha(c) || isdigit(c) || c == '_') {
        t._str += (char)c;
        c = is.get();
    }
    is.unget(), is.clear();
    if (iskeyword(t._str))
        t._what = token::keyword;
    return is;
case EOF:
        while (c != '/' && c != '#' && !strchr("abcdefghimnoprstuvwx", c) &&
            c != '\'' && c != '"' && c != EOF) {
            t._str += (char)c;
            c = is.get();
        }
        is.unget(), is.clear();
        return is;
    }
}
```

The function `html()` replaces the characters '`&`', '`<`', '`>`' and '`"`' in string `s` by its **HTML** equivalents and re
spaces.

```
string html(const string& s)
{
    string s1;
    string::size_type i;
    for (i = 0; i < s.length(); i++) {
        switch (s[i]) {
            case '&':
                s1 += "&";
                break;
            case '<':
                s1 += "<";
                break;
            case '>':
                s1 += ">";
                break;
            case '"':
                s1 += """;
                break;
            case '\t':
                s1.append(tabsize, ' ');
                break;
            default:
                s1 += s[i];
        }
    }
    return s1;
}
```

`Operator<<` sends a token to an output stream. The code is straightforward.

```
ostream& operator<<(ostream& os, const token& t)
{
    if (t._what == token::code)
        cout << html(t._str);
    else if (t._what == token::comment)
        cout << "<span class=comment>" << html(t._str) << "</span>";
    else if (t._what == token::keyword)
        cout << "<span class=keyword>" << html(t._str) << "</span>";
    else if (t._what == token::pp)
        cout << "<span class=pp>" << html(t._str) << "</span>";
    else
        cout << html(t._str);
    return os;
}
```

This is the entry point of *Cpphtml*. All code will be wrapped in a `<pre>` element. By overloading `operator`
the `while` loop is very short and clean. All output is sent to `cout`.

⊟ Collapse

```
int main(int argc, char **argv)
{
    if (argc != 2 && argc != 3) {
        cout << "usage: cpphtml file [tab size]" << endl;
        return 0;
    }
    ifstream is(argv[1]);
    if (!is.good()) {
        cerr << "bad input file" << endl;
        return -1;
    }
    if (argc == 3) {
        t  b  i        t  i(      [2])
    cout << "<html>" << endl
      << "<head>" << endl
      << "<style>" << endl;
    cout << ".keyword{color:rgb(0,0,255);}" << endl;
    cout << ".comment{color:rgb(0,128,0);}" << endl;
    cout << ".pp{color:rgb(0,0,255);}" << endl;
    cout << "</style>" << endl << "<body>" << endl;
    cout << "<pre style=\"font-family:courier;font-size:10pt\">";
    token t;
    while (is >> t) {
        cout << t;
    }
    cout << "</pre>" << "</body>"
         << endl << "</html>" << endl;
    return 0;
}
```