The C++ Programming Language

C++ Tips and Traps

Outline

Tips for C Programmers
C++ Traps and Pitfalls
Efficiency and Performance

Tips for C Programmers

- Use **const** instead of **#define** to declare program constants, *e.g.*,

  – C

    #define PI 3.14159
    #define MAX_INT 0x7FFFFFFF
    #define MAX_UNSIGNED 0xFFFFFFFF

  – C++

    const double PI = 3.14159;
    const int MAX_INT = 0x7FFFFFFF;
    const unsigned MAX_UNSIGNED = 0xFFFFFFFF;

- Names declared with **#define** are untyped and unrestricted in scope

  – In contrast, names declared with **const** are typed and follow C++ scope rules

    * *e.g.*, **const**s have static linkage...

Tips for C Programmers (cont'd)

- Use inline functions and parameterized types instead of preprocessor macros, *e.g.*,
  - C
    - Macros

      ```
      #define MAX(A,B) (((A) >= (B)) ? (A) : (B))
      /* ...*/
      MAX (a++, b++); /* Trouble! */
      ```

    - Using a type as a parameter:

      ```
      #define DECLARE_MAX(TYPE) \
           TYPE MAX (TYPE a, TYPE b) \
           { return a >= b ? a : b; }
      DECLARE_MAX (int)
      DECLARE_MAX (double)
      DECLARE_MAX (char)
      ```

  - C++

    ```
    inline int MAX (int a, int b) {return a >= b ? a : b;}
    /* ...*/
    MAX (a++, b++); /* No problem! */
    template <class T> inline
    MAX (T a, T b) { return a >= b ? a : b; }
    ```

Tips for C Programmers (cont'd)

- Note, there are still some uses for preprocessor, however, *e.g.*,
  - Wrapping headers and commenting out code blocks:

    ```
    #ifndef _FOOBAR_H
    #define _FOOBAR_H
    ...
    #endif
    ```
  - Stringizing and token pasting

    ```
    #define name2(A,B) A##B
    ```
  - File inclusion

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

Tips for C Programmers (cont'd)

- Be careful to distinguish between **int** and **unsigned**

- Unlike C, C++ distinguishes between **int** and **unsigned int**, so be careful when using overloaded functions:

```
#include <iostream.h>
inline void f (int) { cout << "f (int) called\n"; }
inline void f (unsigned) { cout << "f (unsigned) called\n"; }
int main (void) {
    f (1); // calls f (int)
    f (1U); // calls f (unsigned)
}
```

Tips for C Programmers (cont'd)

- Consider using references instead of pointers as function arguments, *e.g.*,

  – C

```
void screen_size (unsigned *height, unsigned *width);
/* ...*/
unsigned height, width;
screen_size (&height, &width);
```

  – C++

```
void screen_size (unsigned &height, unsigned &width);
// ...
unsigned height, width;
screen_size (height, width);
```

- However, it is harder to tell if arguments are modified with this approach!

Tips for C Programmers (cont'd)

- Declare reference or pointer arguments that are not modified by a function as **const**, *e.g.*,

  - C

    struct Big_Struct { **int** array[100000], **int** size; };

    void foo (**struct** Big_Struct *bs);
    // passed as pointer for efficiency

    int strlen (**char** *str);

  - C++

    void foo (**const** Big_Struct &bs);

    int strlen (**const char** *str);

- This allows callers to use **const** values as arguments and also prevents functions from accidentally modifying their arguments

Tips for C Programmers (cont'd)

- Use overloaded function names instead of different function names to distinguish between functions that perform the same operations on different data types:

  - C

    int abs (**int** x);
    double fabs (**double** x);
    long labs (**long** x);

  - C++

    int abs (**int** x);
    double abs (**double** x);
    long abs (**long** x);

- Do not forget that C++ does *NOT* permit overloading on the basis of return type!

Tips for C Programmers (cont'd)

- Use **new** and **delete** instead of `malloc` and `free`, *e.g.*,
  - C

    ```
    int size = 100;
    int *ipa = malloc (size); /* Error!!! */
    /* ...*/
    free (ipa);
    ```

  - C++

    ```
    const int size = 100;
    int *ipa = new int[size];
    // ...
    delete ipa;
    ```

- **new** can both help avoid common errors with `malloc` and also ensure that constructors and destructors are called

Tips for C Programmers (cont'd)

- Use iostream I/O operators << and >> instead of `printf` and `scanf`
  - C

    ```
    float x;
    scanf ("%f", &x);
    printf ("The answer is %f\n", x);
    fprintf (stderr, "Invalid command\n");
    ```

  - C++

    ```
    cin >> x;
    cout << "The answer is " << x << "\n";
    cerr << "Invalid command\n";
    ```

- The << and >> stream I/O operators are (1) type-safe and (2) extensible to user-defined types

Tips for C Programmers (cont'd)

- Use **static** objects with constructor/destructors instead of explicitly calling initialization/finalization functions

  – C

```
struct Symbol_Table {
      /* ...*/
};
void init_symbol_table (struct Symbol_Table *);
int lookup (struct Symbol_Table *);
static struct Symbol_Table sym_tab;
int main (void) {
      char s[100];
      init_symbol_table (&sym_tab);
      /* ...*/
}
```

  – C++

```
class Symbol_Table : private Hash_Table {
public:
      Symbol_Table (void); // init table
      int lookup (String &key);
      ~Symbol_Table (void);
};
static Symbol_Table sym_tab;
int main (void) {
      String s;
      while (cin >> s)
            if (sym_tab.lookup (s) != 0)
                  cout << "found " << s << "\n";
}
```

Tips for C Programmers (cont'd)

- Declare variables near the place where they are used, and initialize variables in their declarations, *e.g.*,
  - C

    ```
    void dup_assign (char **dst, char *src) {
        int len;
        int i;
        if (src == *dst) return;
        if (*dst != 0) free (*dst);
        len = strlen (src);
        *dst = (char *) malloc (len + 1);
        for (i = 0; i < len; i++) (*dst)[i] = src[i];
    }
    ```

  - C++

    ```
    void dup_assign (char *&dst, const char *src) {
        if (src == dst) return;
        delete dst; // delete checks for dst == 0
        int len = strlen (src);
        dst = new char[len + 1];
        for (int i = 0; i < len; i++) dst[i] = src[i];
    }
    ```

Tips for C Programmers (cont'd)

- Use derived classes with virtual functions rather than using **switch** statements on type members:

  - C

```
#include <math.h>
enum Shape_Type {
    TRIANGLE, RECTANGLE, CIRCLE
};

struct Triangle { float x1, y1, x2, y2, x3, y3; };
struct Rectange { float x1, y1, x2, y2; };
struct Circle { float x, y, r; };

struct Shape {
    enum Shape_Type shape;
    union {
        struct Triange t;
        struct Rectange r;
        struct Circle c;
    } u;
};
```

- C (cont'd)

```
float area (struct Shape *s) {
     switch (s->shape) {
     case TRIANGLE:
          struct Triangle *p = &s->u.t;
          return fabs (
               (p->x1 * p->y2 – p->x2 * p->y1) +
               (p->x2 * p->y3 – p->x3 * p->y2) +
               (p->x3 * p->y1 – p->x1 * p->y3)) / 2;
     case RECTANGLE:
          struct Rectange *p = &s->u.r;
          return fabs ((p->x1 – p->x2) *
                    (p->y1 – p->y2));
     case CIRCLE:
          struct Circle *p = &s->u.c;
          return M_PI * p->r * p->r;
     default:
          fprintf (stderr, "Invalid shape\n");
          exit (1);
     }
}
```

- C++

```
#include <iostream.h>
#include <math.h>
class Shape {
public:
     Shape () {}
     virtual float area (void) const = 0;
};
class Triangle : public Shape {
public:
     Triangle (float x1, float x2, float x3,
          float y1, float y2, float y3);
     virtual float area (void) const;
private:
     float x1, y1, x2, y2, x3, y3;
};
float Triangle::area (void) const {
     return fabs ((x1 * y2 - x2 * y1) +
               (x2 * y3 - x3 * y2) +
               (x3 * y1 - x1 * y3)) / 2;
}
```

- C++

```
class Rectange : public Shape {
public:
     Rectangle (float x1, float y1, float x2, float y2);
     virtual float area (void) const;
private:
     float x1, y1, x2, y2;
};
float Rectangle::area (void) const {
     return fabs ((x1 - x2) * (y1 - y2));
}
class Circle : public Shape {
public:
     Circle (float x, float y, float r);
     virtual float area (void) const;
private:
     float x, y, r;
};
float Circle::area (void) const {
     return M_PI * r * r;
}
```

Tips for C Programmers (cont'd)

- Use static member variables and functions instead of global variables and functions, and place **enum** types in class declarations

- This approach avoid polluting the global name space with identifiers, making name conflicts less likely for libraries

  - C

```
#include <stdio.h>
enum Color_Type { RED, GREEN, BLUE };
enum Color_Type color = RED;
unsigned char even_parity (void);
int main (void) {
    color = GREEN;
    printf ("%.2x\n", even_parity ('Z'));
}
```

Tips for C Programmers (cont'd)

- static members (cont'd)
  - C++

```
#include <iostream.h>
class My_Lib {
public:
     enum Color_Type { RED, GREEN, BLUE };
     static Color_Type color;
     static unsigned char even_parity (char c);
};
My_Lib::Color_Type My_Lib::color = My_Lib::RED;
int main (void) {
     My_Lib::color = My_Lib::GREEN;
     cout << hex (int (My_Lib::even_parity ('Z')))
          << "\n";
}
```

- Note that the new C++ "namespaces" feature will help solve this problem even more elegantly

Tips for C Programmers (cont'd)

- Use anonymous unions to eliminate unnecessary identifiers
  - C

```
unsigned hash (double val) {
      static union {
            unsigned asint[2];
            double asdouble;
      } u;
      u.asdouble = val;
      return u.asint[0] ^ u.asint[1];
}
```

  - C++

```
unsigned hash (double val) {
      static union {
            unsigned asint[2];
            double asdouble;
      };
      asdouble = val;
      return asint[0] ^ asint[1];
}
```

C++ Traps and Pitfalls

- Ways to circumvent C++'s protection scheme:

  #define private public
  #define const
  #define class struct

- Note, in the absence of exception handling it is very difficult to deal with constructor failures

  – *e.g.*, in operator overloaded expressions that create temporaries

C++ Traps and Pitfalls (cont'd)

- Initialization vs Assignment
  - Consider the following code

```
class String {
public:
     String (void); // Make a zero-len String
     String (const char *s); // char * --> String
     String (const String &s); // copy constructor
     String &operator= (const String &s); // assignment
private:
     int len;
     char *data;
};
class Name {
public:
     Name (const char *t) { s = t; }
private:
     String s;
};
int main (void) {
     // How expensive is this?????????
     Name neighbor = "Joe";
}
```

- Initialization vs Assignment (cont'd)

  – Constructing "neighbor" object is costly

1. `Name::Name` gets called with parameter "Joe"

2. `Name::Name` has no base initialization list, so member object "'neighbor.s"' is constructed by default `String::String`

   ∗ This will probably allocate a 1 byte area from freestore for the '\0'

3. A temporary "Joe" String is created from parameter t using the `CONST CHAR *` constructor

   ∗ This is another freestore allocation and a `strcpy`

4. `String::operator= (const string &)` is called with the temporary String

5. This will **delete** the old string in `s`, use another **new** to get space for the new string, and do another `strcpy`

6. The temporary String gets destroyed, yet another freestore operation

– Final score: 3 **new**, 2 **`strcpy`**, and 2 **delete**
  Total "cost units": 7

- Initialization vs Assignment (cont'd)

  – Compare this to an initialization-list version. Simply replace

    Name::Name (**const char*** t) { s = t; }
    with
    Name::Name (**const char*** t): s (t) { }

  – Now construction of "neighbor" is:

    1. Name::Name (**const char** *) gets called with parameter "Joe"

    2. Name::Name (**const char** *) *has* an init list, so **`neighbor::s`** is initialized from `S` with String::String (**const char** *)

    3. String::String ("Joe") will probably do a **new** and a **`strcpy`**

  – Final score: 1 **new**, 1 **`strcpy`**, and 0 **delete**
    Total "cost units": 2

  – Conclusion: *always* use the initialization syntax, even when it does not matter...

C++ Traps and Pitfalls (cont'd)

- Although a function with no arguments must be called with empty parens a constructor with no arguments must be called with *no* parens!

```
class Foo {
public:
     Foo (void);
     int bar (void);
};
int main (void) {
     Foo f;
     Foo ff (); // declares a function returning Foo!
     f.bar (); // call method
     f.bar; // a no-op
     ff.bar (); // error!
}
```

C++ Traps and Pitfalls (cont'd)

- Default Parameters and Virtual Functions

```
extern "C" int printf (const char *, ...);

class Base {
public:
     virtual void f (char *name = "Base") {
          printf ("base = %s\n", name);
     }
};

class Derived : public Base {
public:
     virtual void f (char *name = "Derived") {
          printf ("derived = %s\n", name);
     }
};

int main (void) {
     Derived *dp = new Derived;
     dp->f (); /* prints "derived = Derived" */

     Base *bp = dp;
     bp->f (); /* prints "derived = Base" */
     return 0;
}
```

C++ Traps and Pitfalls (cont'd)

- Beware of subtle whitespace issues...

```
int b = a //* divided by 4 */4;
-a;
/* C++ preprocessing and parsing */
int b = a -a;
/* C preprocessing and parsing */
int b = a/4; -a;
```

- Note, in general it is best to use whitespace around operators and other syntactic elements, *e.g.*,

```
char *x;
int foo (char * = x); // OK
int bar (char*=x); // Error
```

Efficiency and Performance

- *Inline Functions*
  - Use of inlines in small programs can help performance, extensive use of inlines in large projects can actually hurt performance by enlarging code, bringing on paging problems, and forcing many recompilations
  - Sometimes it's good practice to turn-off inlining to set a worst case performance base for your application, then go back an inline as part of performance tuning

- *Parameter Passing*
  - Passing C++ objects by reference instead of value is a good practice
    * It's rarely to your advantage to replicate data and fire off constructors and destructors unnecessarily

Efficiency and Performance (cont'd)

- *Miscellaneous Tips*
  - Use good memory (heap) management strategies
  - Develop good utility classes (for strings, in particular)
  - Good object and protocol design (particularly, really isolating large-grained objects)
  - Give attention to paging and other ways your application uses system resources

- While C++ features, if used unwisely, can slow an application down, C++ is not inherently slower than say C, particularly for large scale projects
  - In fact, as the size and complexity of software increases, such comparisons aren't evenrelevant since C fails to be a practical approach whereas C++ comes into its own