

# Low Level Types and Operations

## Version 1.3.1

Andrew Pochinsky

November 10, 2003

## 1 QCD HEADER FILE

Collect all implementations into one header file so that we don't need to change sources in order to try a new approach.

```
<qcd.hh>≡
    #ifndef QCD_HH
    #define QCD_HH
    <Implementation>
    #endif
```

## 2 VANILLA C

Here is a header for plain C implementation. There is no need to bother with actual working code, we only want to provide correct data sizes and enough prototypes to be able to compile applications on gcc 2.9.

```
<Implementation>≡
    #ifdef USE_C
    #include "qcd-c.hh"
    #endif

<qcd-c.hh>≡
    #ifndef QCD_C_HH
    #define QCD_C_HH
    <C++ real datatypes>
    <C++ real functions>
    <C++ complex datatypes>
    <C++ complex functions>
    #endif
```

First, we need a scalar `complex` — just to avoid remembering which part of an array is real and which imaginary.

```
<C++ complex datatypes>≡
    struct complex {
        real re, im;
        <C++ complex constructors>
    };
```

Now let us turn to a C++ implementation of `vcomplex` type. For the sake of efficiency, we define all functions dealing with it online and implement then in terms of operations on `vreal`. Such an approach allows use to reuse `vcomplex` implementation if a vector architecture lacks complex number oriented instructions.

```
<C++ complex datatypes>+≡
    struct vcomplex {
        vreal re, im;
        <C++ vcomplex constructors>
        <C++ vcomplex methods>
    };
```

Here is a collection of `vcomplex` constructors that DWF's need:

```
<C++ vcomplex constructors>≡
    inline vcomplex(): re(), im() {}
    inline vcomplex(real r, real i): re(r), im(i) {}
    inline vcomplex(const vreal &r, const vreal &i): re(r), im(i) {}
```

```

{C++ complex constructors}≡
    inline complex(): re(), im() {}
    inline complex(real r, real i): re(r), im(i) {}

```

Some of operations on `vcomplex` look better when written as class methods. There is no intrinsic value in such an approach, it simply looks nice.

```

{C++ vcomplex methods}≡
    inline void vput_0(const complex &v) { re.vput_0(v.re); im.vput_0(v.im); }
    inline void vput_1(const complex &v) { re.vput_1(v.re); im.vput_1(v.im); }
    inline void vput_2(const complex &v) { re.vput_2(v.re); im.vput_2(v.im); }
    inline void vput_3(const complex &v) { re.vput_3(v.re); im.vput_3(v.im); }

    complex sum() const { return complex(re.sum(), im.sum()); }

    vcomplex shift_down1(const vcomplex &b) const {
        return vcomplex(re.shift_down1(b.re), im.shift_down1(b.im));
    }
    vcomplex shift_down2(const vcomplex &b) const {
        return vcomplex(re.shift_down2(b.re), im.shift_down2(b.im));
    }
    vcomplex shift_down3(const vcomplex &b) const {
        return vcomplex(re.shift_down3(b.re), im.shift_down3(b.im));
    }

    vcomplex shift_up1(const vcomplex &b) const {
        return vcomplex(re.shift_up1(b.re), im.shift_up1(b.im));
    }
    vcomplex shift_up2(const vcomplex &b) const {
        return vcomplex(re.shift_up2(b.re), im.shift_up2(b.im));
    }
    vcomplex shift_up3(const vcomplex &b) const {
        return vcomplex(re.shift_up3(b.re), im.shift_up3(b.im));
    }

```

Other complex operations are implemented as overloaded functions. There are schools of design saying overloading is evil, but C++ uses it heavily, and we can not save the world by avoiding this particular misfeature of the language.

First, the ring operations:

```

{C++ complex functions}≡
    inline vcomplex operator+(const vcomplex &a, const vcomplex &b) {
        return vcomplex(a.re + b.re, a.im + b.im);
    }
    inline vcomplex operator-(const vcomplex &a, const vcomplex &b) {
        return vcomplex(a.re - b.re, a.im - b.im);
    }
    inline vcomplex operator*(const vcomplex &a, const vcomplex &b) {
        return vcomplex(a.re * b.re - a.im * b.im, a.im * b.re + a.re * b.im);
    }
    inline vcomplex operator*(const vreal &a, const vcomplex &b) {
        return vcomplex(a * b.re, a * b.im);
    }

```

Application of  $\gamma$ -matrices to fermions is greatly accelerated by computation of expressions of the form  $a \pm b$ ,  $a \pm ib$ ,  $a \leftarrow a \pm b$  and  $a \leftarrow a \pm ib$ :

```

<C++ complex functions>+=
    inline vcomplex ap1b(const vcomplex &a, const vcomplex &b) {
        return a + b;
    }
    inline vcomplex am1b(const vcomplex &a, const vcomplex &b) {
        return a - b;
    }
    inline vcomplex apib(const vcomplex &a, const vcomplex &b) {
        return vcomplex(a.re - b.im, a.im + b.re);
    }
    inline vcomplex amib(const vcomplex &a, const vcomplex &b) {
        return vcomplex(a.re + b.im, a.im - b.re);
    }

    inline void set_ap1b(vcomplex &a, const vcomplex &b) { a = ap1b(a,b); }
    inline void set_am1b(vcomplex &a, const vcomplex &b) { a = am1b(a,b); }
    inline void set_apib(vcomplex &a, const vcomplex &b) { a = apib(a,b); }
    inline void set_amib(vcomplex &a, const vcomplex &b) { a = amib(a,b); }

```

## 2.1 Placeholder for vreal type

There is no immediate need to have a plain C++ implementation of the vector operations. Hence, we only provide enough prototypes to allow for compilation of the DWF code on gcc 2.9.x.

For orthogonality, let us start with `real` alias

```

<C++ real datatypes>+=
    typedef float real;

```

About the only thing one can tell about `vreal` is that it is four `real` long:

```

<C++ real datatypes>+=
    struct vreal {
        real v[4];
        <C++ vreal constructors>
        <C++ vreal methods>
    };

```

In the following, assume that `vreal` contains four `real` numbers which we denote as  $\mathbf{a} = [a_0, a_1, a_2, a_3]$ . We specifically do not reveal representation details of memory layout to allow greater flexibility of implementation.

One needs to provide several constructors for `vreal`. First, uninitialized vector:

```

<C++ vreal constructors>+=
    vreal();

```

Next, replicating a scalar value in all four pieces of `vreal`, e.g.,  $\mathbf{a} = [v, v, v, v]$ :

```

<C++ vreal constructors>+=
    vreal(real v);

```

Also, assembling four separate `real`'s into one `vreal`, e.g.,  $\mathbf{a} = [v_0, v_1, v_2, v_3]$ :

```

<C++ vreal constructors>+=
    vreal(real v0, real v1, real v2, real v3);

```

Finally, we also need a copy constructor because gcc does sometimes funny things without it:

```

<C++ vreal constructors>+=
    vreal(const vreal &b);

```

Four modifiers for individual components of `vreal` follow:

```

<C++ vreal methods>+=
    void vput_0(real);
    void vput_1(real);
    void vput_2(real);
    void vput_3(real);

```

Next we need to define six shift operations. In the above notation they are

$$\begin{aligned}
a.\text{shift\_down1}(b) &= [a_3, b_0, b_1, b_2] \\
a.\text{shift\_down2}(b) &= [a_2, a_3, b_0, b_1] \\
a.\text{shift\_down3}(b) &= [a_1, a_2, a_3, b_0] \\
a.\text{shift\_up1}(b) &= [a_1, a_2, a_3, b_0] \\
a.\text{shift\_up2}(b) &= [a_2, a_3, b_0, b_1] \\
a.\text{shift\_up3}(b) &= [a_3, b_0, b_1, b_2]
\end{aligned}$$

```

<C++ vreal methods>+=
    vreal shift_up1(const vreal &b) const;
    vreal shift_up2(const vreal &b) const;
    vreal shift_up3(const vreal &b) const;
    vreal shift_down1(const vreal &b) const;
    vreal shift_down2(const vreal &b) const;
    vreal shift_down3(const vreal &b) const;

```

The last method we need computes a sum of all components of `vreal`:

```

<C++ vreal methods>+=
    real sum(void) const;

```

Other operations on `vreal` are defined as C++ operators:

```

<C++ real functions>=
    vreal operator+(const vreal &, const vreal &);
    vreal operator-(const vreal &, const vreal &);
    vreal operator*(const vreal &, const vreal &);

```

### 3 GCC SSE IMPLEMENTATION

Starting with some 3.x.x version, gcc supports SSE datatypes and some related operations on Pentium 4. Here is an implementation of `vreal` for it.

```

<Implementation>+=
    #ifdef USE_GCC_SSE
    #include "qcd-gcc-sse.hh"
    #endif

<qcd-gcc-sse.hh>=
    #ifndef _QCD_GCC_SSE_HH
    #define _QCD_GCC_SSE_HH
    <GCC SSE real datatypes>
    <GCC SSE real functions>
    <C++ complex datatypes>
    <C++ complex functions>
    #endif

<GCC SSE real datatypes>=
    typedef float real;
    typedef float VREAL __attribute__((mode(V4SF),aligned(16)));

<GCC SSE real datatypes>+=
    struct vreal {
        VREAL v;
        <GCC SSE vreal constructors>
        <GCC SSE vreal methods>
    };

<GCC SSE vreal constructors>=
    inline vreal() {}

```

```

<GCC SSE vreal constructors>+≡
inline vreal(const real &a) {
    v = __builtin_ia32_loadss((float *)&a);
    asm("shufps\t$0,%0,%0" : "+x" (v));
}

<GCC SSE vreal constructors>+≡
inline vreal(real a0, real a1, real a2, real a3) {
    vput_0(a0); vput_1(a1); vput_2(a2); vput_3(a3);
}

<GCC SSE vreal constructors>+≡
vreal(const vreal &b): v(b.v) {}

<GCC SSE vreal methods>≡
inline void vput_0(real a) { ((real *)&v)[0] = a; }
inline void vput_1(real a) { ((real *)&v)[1] = a; }
inline void vput_2(real a) { ((real *)&v)[2] = a; }
inline void vput_3(real a) { ((real *)&v)[3] = a; }

<GCC SSE vreal methods>+≡
inline vreal shift_up1(const vreal &b) const {
    vreal x = *this;
    vreal y = b;
    asm("shufps\t$0x30,%0,%1\n\t"
        "shufps\t$0x29,%1,%0"
        : "+x" (x.v), "+x" (y.v));
    return x;
}

inline vreal shift_up2(const vreal &b) const {
    vreal x = *this;
    asm("shufps\t$0x4e,%1,%0"
        : "+x" (x.v): "x" (b.v));
    return x;
}

inline vreal shift_up3(const vreal &b) const {
    vreal x = *this;
    asm("shufps\t$0x03,%1,%0\n\t"
        "shufps\t$0x9c,%1,%0"
        : "+x" (x.v): "x" (b.v));
    return x;
}

inline vreal shift_down1(const vreal &b) const {
    return shift_up3(b);
}

inline vreal shift_down2(const vreal &b) const {
    return shift_up2(b);
}

inline vreal shift_down3(const vreal &b) const {
    return shift_up1(b);
}

<GCC SSE vreal methods>+≡
inline real sum(void) const {
    real *vv = (real *)&v;
    return vv[0] + vv[1] + vv[2] + vv[3];
}

```

```

<GCC SSE real functions>≡
inline vreal
operator+(const vreal &a, const vreal &b) {
    vreal r;
    r.v = __builtin_ia32_addps(a.v, b.v);
    return r;
}
inline vreal
operator-(const vreal &a, const vreal &b) {
    vreal r;
    r.v = __builtin_ia32_subps(a.v, b.v);
    return r;
}
inline vreal
operator*(const vreal &a, const vreal &b) {
    vreal r;
    r.v = __builtin_ia32_mulps(a.v, b.v);
    return r;
}

```