μC++ to C∀ Cheat Sheet

1 Introduction

C∀ is NOT an object-oriented programming-language. C∀ uses parametric polymorphism and allows overloading of variables and routines:

```
int i;  char i;  double i;        // overload name i
int i();  double i();  char i();
i += 1;                  // int i
i += 1.0;                // double i
i += 'a';                // char i
int j = i();             // int i()
double j = i();          // double i();
char j = i();            // char i()
```

C∀ has rebindable references.

```
int x = 1, y = 2, * p1x = &x, * p1y = &y, ** p2i = &p1x,
                  && r1x = x, & r1y = y, && r2i = r1x;
```

`**p2i = 3;`	`r2i = 3;  // change x`
`p2i = &p1y;`	`&r2i = &r1y;  // change p2i / r2i`
`**p2i = 3;`	`r2i = 3;  // change y`
`p1x = p1y;`	`&r1x = &r1y;  // change p1x / r1x`
`**p2i = 4;`	`r2i = 4;  // change y`
`p1x = nullptr;`	`&r1x = 0p;  // reset`

Non-rebindable reference (C++ reference) is a **const** reference (**const** pointer).

```
int & const cr = x; // must initialize, no null pointer
int & const & const crcr = cr; // generalize
```

Aggregate qualification is reduced or eliminated by opening scopes using the **with** clause.

```
struct S { int i; int j; double m; };  // field i has same type in structures S and T
struct T { int i; int k; int m; };
void foo( S s, T t ) with(s, t) {   // open structure scope s and t in parallel
    j + k;                  // unambiguous, s.j + t.k
    m = 5.0;                // unambiguous, s.m = 5.0
    m = 1;                  // unambiguous, t.m = 1
    int a = m;              // unambiguous, a = t.m
    double b = m;           // unambiguous, b = s.m
    int c = s.i + t.i;      // unambiguous with qualification
    (double)m;              // unambiguous with cast s.m
}
```

In subsequent code examples, the left example is μC++ and the right example is C∀.

2 Stream I/O

C∀ output streams automatically separate values and insert a newline at the end of the print.

```
#include <iostream>
using namespace std;
int i;  double d;  char c;
cin >> i >> d >> c;
cout << i << ' ' << d << ' ' << c | endl;
```

```
#include <fstream.hfa>

int i;  double d;  char c;
sin | i | d | c;
sout | i | d | c
```

3 Looping

```
for ( ;; ) { ... }  /  while ( true ) { ... }
for ( int i = 0; i < 10; i += 1 ) { ... }
for ( int i = 5; i < 15; i += 2 ) { ... }
for ( int i = -1; i <= 10; i += 3 ) { ... }
for ( int i = 10; i > 0; i -= 1 ) { ... }
```

```
for () { ... }  /  while () { ... }
for ( 10 ) { ... }  /  for ( i; 10 ) { ... }
for ( i; 5 ~ 15 ~ 2 ) { ... }
for ( i; -1 ~= 10 ~ 3 ) { ... }
for ( i; 0 -~ 10 ) { ... }
```

```
int i = 0
for ( i = 0; i < 10; i += 1 ) { ... }
if ( i == 10 ) { ... }
```

```
for ( i; 10 ) { ... }
else { ... } // i == 10
```

```
L1: for ( ;; ) {
    L2: for ( ;; ) {
        ... break L1; ... break L2; ...
    }
}
```

```
L1: for () {
    L2: for () {
        ... break L1; ... break L2; ...
    }
}
```

4 Exception

Currently, C∀ uses macros ExceptionDecl and ExceptionInst to declare and instantiate an exception.

```
_Exception E { // local or global scope
    ... // exception fields
};
try {
    ...
    if ( ... ) _Resume E( /* initialization */ );
    if ( ... ) _Throw E( /* initialization */ );
        ...
} _CatchResume( E & ) { // should be reference
    ...
} catch( E & ) {
    ...
}
```

```
#include <Exception.hfa>
ExceptionDecl( E,       // must be global scope
    ... // exception fields
);
try {
    ...
    if ( ... ) throwResume ExceptionInst( E, /* intialization */ );
    if ( ... ) throw ExceptionInst( E, /* intialization */ );
    ...
} catchResume( E * ) { // must be pointer
    ...
} catch( E * ) {
    ...
}
```

5 Non-local Exception

```
void main() {
    try {
        _Enable {
            ... suspend(); ...
        }
    } _CatchResume( E & ) { // reference
        ...
    } catch( E & ) {
        ...
    }
}
```

```
#define resumePoll( coroutine ) resume( coroutine ); checked_poll()
#define suspendPoll suspend; checked_poll()
void main() {
    try {
        enable_ehm();
        ... suspendPoll ...
        disable_ehm();
    } catchResume( E * ) { // pointer
        ...
    } catch( E & ) {
        ...
    }
}
```

6 Constructor / Destructor

```
struct S {
    ... // fields
    S(...) { ... }
    ~S(...) { ... }
};
```

```
struct S {
    ... // fields
};
?{}( S & s, ...) { ... }
^?{}( S & s ) { ... }
```

7 String

```
string s1, s2;
s1 = "hi";
s2 = s1;
s1 += s2;
s1 == s2; s1 != s2;
s1 < s2; s1 <= s2; s1 > s2; s1 >= s2;
s1.length();
s1[3];
s1.substr( 2 ); s1.substr( 2, 3 );
s1.replace( 2, 5, s2 );
s1.find( s2 ), s1.rfind( s2 );
s1.find_first_of( s2 ); s1.find_last_of( s2 );
s1.find_first_not_of(s2 ); s1.find_last_not_of( s2 );
getline( cin, s1 );
cout << s1 << endl;
```

```
s1 = "hi";
s2 = s1;
s1 += s2;
s1 == s2; s1 != s2;
s1 < s2; s1 <= s2; s1 > s2; s1 >= s2;
size( s1 );
s1[3];
s1( 2 ); s1( 2, 3 );
//s1.replace( 2, 5, s2 );
find( s1, s2 ), rfind( s1, s2 );
find_first_of( .substr, s2 ); s1.find_last_of( s2 );
s1.find_first_not_of(s2 ); s1.find_last_not_of( s2 );
sin | getline( s1 );
sout | s1;
```

8 Structures (object-oriented *vs.* routine style)

```
struct S {
    int i = 0;
    int setter( int j ) { int t = i; i = j; return t; }
    int getter() { return i; }
};

S s;
s.setter( 3 );  // object-oriented call
int k = s.getter();
```

```
struct S {
    int i;
};
void ?{}( S & s ) { s.i = 0; }
int setter( S & s, int j ) with(s) { int t = i; i = j; return t; }
int getter( S & s ) with(s) { return i; }
S s;
setter( s, 3 );  // normal routine call
int k = getter( s );
```

9 uNoCtor

```
struct S {
    int i;
    S( int i ) { S::i = i; cout << S::i << endl; }
};
uNoCtor<S> s[10];
int main() {
    for ( int i = 0; i < 10; i += 1 ) {
        s[i].ctor( i );
    }
    for ( int i = 0; i < 10; i += 1 ) {
        cout << s[i]->i << endl;
    }
}
```

```
struct S {
    int i;
};
void ?{}( S & s, int i ) { s.i = i; sout | s.i; }
S s[10] @= {};
int main() {
    for ( i; 10 ) {
        ?{}( s[i], i );  // call constructor
    }
    for ( i; 10 ) {
        sout | s[i].i; // dot not arrow
    }
}
```

10 Coroutines

```
_Coroutine C {
    // private coroutine fields
    void main() {
        ... suspend(); ...
        ... _Resume E( ... ) _At partner;
    }
  public:
    void mem( ... ) {
        ... resume() ...
    }
};
C c;
```

```
#include <coroutine.hfa>
coroutine C {
    // private coroutine fields

};
void main( C & c ) {
    ... suspend; ... // keyword not routine
    ... resumeAt( partner, ExceptionInst( E, ... ) );
}
void mem( C & c, ... ) {
    ... resume(); ...
}
```

11 Locks

```
uOwnerLock m;
uCondLock s;
bool avail = true;
m.acquire();
if ( ! avail ) s.wait( m );
else {
    avail = false;
    m.release();
}
osacquire( cout ) << i << endl;
```

```
#include <locks.hfa>
owner_lock m;
condition_variable( owner_lock ) s;
bool avail = true;
lock( m );
if ( ! avail ) wait( s, m );
else {
    avail = false;
    unlock( m );
}
mutex( sout ) sout | i;  // safe I/O
```

12 Monitors

```
_Monitor M {
    uCondition c;
    bool avail = true;
  public:

    void rtn() {
        if ( ! avail ) c.wait();
        else avail = false;
    }
};
M m;
```

```
#include <monitor.hfa>
monitor M {
    condition c;
    bool avail;
};
void ?{}( M & m ) { m.avail = true; }
void rtn( M & m ) with( m ) {
    if ( ! avail ) wait( c );
    else avail = false;
}
```

13 Threads

```
_Task T {
    // private task fields
    void main() {
        ... _Resume E( ... ) _At partner;
    }
  public:
};
T t; // start thread in main routine
```

```
#include <thread.hfa>
thread T {
    // private task fields

};
void main( T & t ) {
    ... resumeAt( partner, ExceptionInst( E, ... ) );
}
```