Operators

Most C++ operators are identical to the corresponding Java operators:

Arithmetic: `* / % + -`

Relational: `< <= = >= > !=`

Logical: `! && ||`

Bitwise: `&` bitwise and; `^` bitwise exclusive or; `|` bitwise inclusive or, `~` complement
<< left shift; >> right shift (implementation defined how the sign bit is handled, there is no >>> in C++)

Special: `? :`
`+= -= ...`
`++x x++ --x x--`

The Arrow Operator

To access a member in an object the dot operator is used. If you have a pointer to the object you use the arrow operator.

```
Point p1(1, 2);
Point* p = new Point(1, 2);
double xLoc1 = p1.getX();
double xLoc2 = p->getX();
```

Instead of using the arrow operator you could first dereference the pointer to get the object pointed to, then use the dot operator: `double xLoc2 = (*p).getX()`. This is difficult to read and shouldn't be used.

The sizeof Operator

`sizeof(type name)` returns the size of an object of the type, in character units. `sizeof(expression)` returns the size of the result type of the expression. Compare:

```
int a[5];
cout << sizeof(int) << endl; // 4 (if an int has 4 bytes)
cout << sizeof(a) << endl;   // 20, 4*5

int* pa = new int[5];
cout << sizeof(pa) << endl;  // 8 (the size of the pointer)
cout << sizeof(*pa) << endl; // 4 (points to int)

vector<int> v = { 1, 2, 3, 4, 5 };
cout << sizeof(v) << endl;   // 24 (implementation defined,
                             // size of the object)
```

`sizeof` cannot be used to retrieve the number of elements of a heap-allocated array or of a vector. (But can use `vector::size()`.)

Type Conversions

In Java, there are implicit type conversions from "small" types to "large" types. To convert in the other direction, where precision is lost, an explicit cast is necessary.

```
double d = 35;   // d == 35.0; implicit
d = 35.67;
int x = (int) d; // x == 35; explicit
```

In C++, there are implicit conversions in both directions.

```
d = 35.67;
int x = d;       // x == 35; implicit
```

You should use an explicit cast instead:

```
int x = static_cast<int>(d);
```

Different Casts

The C++ rules for implicit type conversions are complex, so it is best to always use explicit casts. There are other casts besides `static_cast`:

`dynamic_cast<type>(pointer)` for "downcasting" in an inheritance hierarchy.

`const_cast<type>(variable)` removes "constness" from a variable. Not often used.

`reinterpret_cast<type>(expr)` converts anything to anything else, just reinterprets a bit pattern. Only used in low-level systems programming.

The C (and Java) style of casting, `(int) d`, is also allowed in C++ but should *not* be used; depending on its argument it functions as a `const_cast`, `static_cast`, or a `reinterpret_cast`.

Numeric Conversions

A string of digits cannot be converted to a binary number with a cast, nor can a binary number be converted to a string. In the old standard, "string streams" were used for this purpose:

```
string s1 = "123.45";
istringstream iss(s1); // similar to a Java Scanner
double nbr;
iss >> nbr;
```

The new standard introduced conversion functions C++11:

```
string s1 = "123.45";
double nbr = stod(s1); // "string to double"
...
string s2 = to_string(nbr + 1);
```

Statements

No surprises here:

- `if`, `switch`, `while`, `do while`, `for`.
- The new standard has range-based `for` C++11.
- `break` and `continue`.
- Labeled `break` and `continue` are not available in C++ (you can use `goto` instead, but don't).

Exceptions

Throwing and catching exceptions:

- Anything can be thrown as an exception: an object, a pointer, an `int`, ..., but you usually throw an object (*not* a pointer to an object).
- There are standard exception classes: `exception`, `runtime_error`, `range_error`, `logic_error`, ...
- In an exception handler, the exception object should be passed by reference (`catch (exception& e)`).
- If an exception isn't caught, the library function `terminate` is called. It aborts the program.
- No Java-style "throws"-specification.

Exceptions, example

```
throw runtime_error("Wrong parameter values!");
throw 20;
```

```
try {
   // program statements
} catch (runtime_error& err) {
   // error handling code
} catch (...) {
   // default error handling code
}
```

Functions

Functions in C++ are similar to Java methods, but C++ functions may also be defined outside classes ("free", "stand-alone", or "global", functions).

Functions may be defined as `inline`. It means that calls of the function are replaced with the body of the function.

```
inline int sum(int i, int j) {
    return i + j;
}
```

`inline` is just a request to the compiler to generate inline code — the compiler may choose to ignore it. Inline function definitions must be placed in header files.

constexpr Functions

In some cases, for example when specifying array bounds, a constant expression is necessary. A function that is specified constexpr is evaluated during compilation if possible, and then it generates a constant expression. Example:

```
constexpr int sum(int i, int j) {
    return i + j;
}

void f(int n) {
    int x[sum(4, 5)]; // correct, equivalent to x[9]
    int y[sum(n, 5)]; // wrong, array bound isn't constant
}
```

constexpr functions are implicitly inline.

Local Static Variables

A class attribute in C++ (as in Java) may be `static`. This means that there is only one variable, which is shared between all objects of the class.

In C++, also functions may have static variables. Such a variable is created on the first call of the function, and its value is remembered between successive calls.

Count the number of calls of a function:

```
void f() {
    static int counter = 0;
    counter++;
    ...
}
```

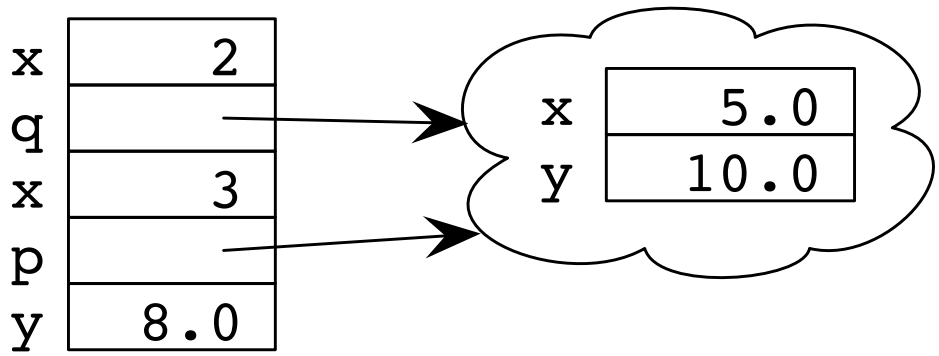
In C, local static variables are essential. In C++, most functions are class members and member variables are used to remember values between function calls.

Argument Passing

Arguments to functions are by default passed by value, as in Java. This means that the value of the argument is computed and copied to the function, and that an assignment to the formal parameter does not affect the argument.

```
void f(int x, const Point* p) {
    x++;
    double y = x + p->getX();
}

int main() {
   int x = 2;
   Point* q = new Point(5, 10);
   f(x, q);
   ...
   delete q;
}
```

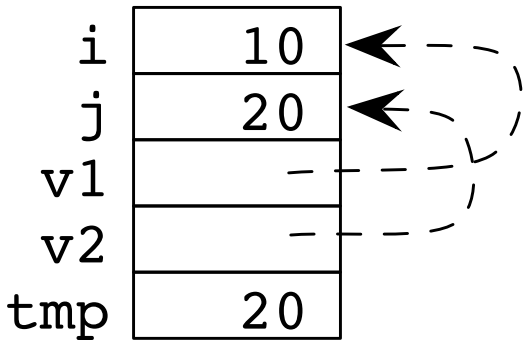


Reference Parameters

If you wish to change the value of an argument you must pass the argument by reference. Example:

```
void swap(int& v1, int& v2) {
    int tmp = v2;
    // see figure for memory state here
    v2 = v1;
    v1 = tmp;
}

int main() {
    int i = 10;
    int j = 20;
    swap(i, j);
}
```



How They Do It in C

C doesn't have references, so to write the `swap` function in C you must use pointers. This should be avoided in C++.

```
void swap(int* v1, int* v2) {
    int tmp = *v2;
    *v2 = *v1;
    *v1 = tmp;
}

int main() {
    int i = 10;
    int j = 20;
    swap(&i, &j);
}
```

Constant Pass by Reference

Pass by reference is also used when you have a large object that you don't wish to copy on each call.

```
bool isShorter(const string& s1, const string& s2) {
    return s1.size() < s2.size();
}
```

Note that `const` is essential (if you don't wish to modify the object):

- Without `const`, you could not pass a constant string as argument, or an object needing conversion (for example `isShorter(s, "abc")`).
- Without `const`, you could inadvertently change the state of the object in the function.

Parameters — Pointer or Reference

If a program uses stack-allocated objects, which should be the normal case, write functions with reference parameters. If the program must use heap-allocated objects, write functions with pointer parameters.

```
void refPrint(const Point& p) {
    cout << p.getX() << " " << p.getY() << endl;
}

void ptrPrint(const Point* p) {
    cout << p->getX() << " " << p->getY() << endl;
}

void f() {
    Point p1(10, 20);
    Point* pptr = new Point(30, 40);
    refPrint(p1);    // normal usage
    ptrPrint(pptr);  // also normal usage
    ...
    delete pptr;
}
```

Pointer or Reference, cont'd

You *can* pass an object even if you have a pointer, or a pointer if you have an object, but that should be avoided:

```
void refPrint(const Point& p) { ... }

void ptrPrint(const Point* p) { ... }

void f() {
    Point p1(10, 20);
    Point* pptr = new Point(30, 40);
    refPrint(*pptr); // possible, but avoid
    ptrPrint(&p1);   // also possible, also avoid
    ...
    delete pptr;
}
```

Array Parameters

An array argument is passed as a pointer to the first element of the array. The size of the array must also be passed as an argument.

```
void print(const int* ia, size_t n) {
    for (size_t i = 0; i != n; ++i) {
        cout << ia[i] << endl;
    }
}

int main() {
    int[] a = {1, 3, 5, 7, 9};
    size_t size = sizeof(a) / sizeof(*a);
    print(a, size);
}
```

Note that `const int*` is a variable pointer to a constant integer. A constant pointer to a variable integer is written `int* const`.

Another Way To Pass Array Parameters

Another way of passing array arguments is inspired by iterators. You pass pointers to the first and one past the last element:

```
void print(const int* beg, const int* end) {
    for (; beg != end; ++beg) {
        cout << *beg << endl;
    }
}

int main() {
    int[] a = {1, 3, 5, 7, 9};
    size_t size = sizeof(a) / sizeof(*a);
    print(a, a + size); // or print(begin(a), end(a))
}
```

Functions with Varying Parameters C++11

A function with an unknown number of arguments of the same type can pack the arguments in an `initializer_list` object. An `initializer_list` has `begin()` and `end()` operations. Example:

```
class MyVector {
public:
    MyVector(const initializer_list<int>& il)
            : v(new int[il.size()]), size(il.size()) {
        size_t i = 0;
        for (auto x : il) { v[i++] = x; }
    }
private:
    int* v;
    int size;
};

MyVector v1({1, 2, 3, 4});
MyVector v2 = {5, 6, 7, 8};
```

Functions Returning Values

The value that a function returns is copied from the function to a temporary object at the call site.

```
string strip(const string& s) {
    string ret;
    for (char c : s) {
        if (c != ' ') {
            ret += c;
        }
    }
    return ret; // the return value is copied
}

int main() {
    string s = strip("Mary had a little lamb") + ".";
    ...
}
```

Functions Returning References

A function can return a reference, and if it does, it can be called on either side of a = operator. This is mainly used when writing operator functions that overload standard operators, e.g., assignment or subscripting. The following example would have made more sense if the array had been a class attribute and `element` had been an overloaded `operator[]`.

```
int& element(int* x, size_t i) {
    // ... could insert index check here
    return x[i];
}

void f() {
    int squares[] = {1, 4, 9, 15, 25};
    element(squares, 3) = 16;
    ...
}
```

Reference and Pointer Pitfalls

Never return a reference or a pointer to a local object. The return value will refer to memory that has been deallocated and will be reused on the next function call.

```
const string& strip(const string& s) { // note & on the return type
    string ret;
    ...
    return ret; // a reference to the local variable is copied
}

int main() {
    string s = strip("Mary had a little lamb") + ".";
    ... disaster
}
```