SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220420<A|D>

Replace <A|D> with this section's letter

Generic programming II

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- In-class problem
- `utility` library

In-class problem

Implementing iterators

```
class Vector {
    ... // Vector implementation
};

int main() {
    Vector vec;

    vec.push_back(17);
    vec.push_back(42);
    vec.push_back(6);
    vec.push_back(28);

    for (size_t i = 0; i < vec.size(); ++i) {
        cout << vec[i] << ' ';
    }

    cout << endl;
}
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
17 42 6 28
```

Implementing iterators

```
class Vector {
    ... // Vector implementation
};

int main() {

    ...
    vec[0] = 100;
    Vector vec2 = vec;
    for (size_t i = 0; i < vec2.size(); ++i) {
        cout << vec2[i] << ' ';
    }
    cout << endl;
}
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
17 42 6 28
100 42 6 28
```

Implementing iterators

```
class Vector {
    ... // Vector implementation
};

int main() {
    ...

    for (~~size_t i = 0; i < vec2.size(); ++i)~~ {
        cout << ~~vec2[i~~] << ' ';
    }

    cout << endl;

}
```

Implementing iterators

```
class Vector {
    ... // Vector implementation
};

int main() {
    ...

    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << ~~vec2[i~~] << ' ';
    }

    cout << endl;
}
```

Implementing iterators

```
class Vector {
    ... // Vector implementation
};

int main() {
    ...

    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
17 42 6 28
100 42 6 28
```

Implementing iterators

```
class Vector {
    // implement iterators
    ... // Vector implementation
};
```

Implementing iterators

```
class Vector {
    class Iterator {
    public:
         Iterator(int* ptr = nullptr) : ptr(ptr) {}
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

← "points to" int in Vector

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        ___ operator++() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

only implementing pre-increment

* ranged for requires pre-increment

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        _1_ operator++() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220420<A|D>

Replace <A|D> with this section's letter

Which type replaces blank #1 for implementing pre-increment `operator++()`?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        _1_ operator++() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            // advance ptr
            ---
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            // advance ptr
            _2_
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- `++` operator
- `*` operator
- `!=` operator

Which expression replaces blank #2 to advance `ptr` to "point to" the next `int` in the `Vector`?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            // advance ptr
            _2_
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            // advance ptr
            ++ptr;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return ___;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return _3_;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Which expression replaces blank #3 to return the current `Iterator` object?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return _3_;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ++ operator
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        ___ operator*() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        _4_ operator*() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Which type replaces blank #4 so that the value "pointed to" by the `Iterator` can be modified when returned?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        _4_ operator*() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() { }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() { ___ }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() { _5_ }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Which statement replaces blank #5 to return the value currently pointed to by the `Iterator`?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() { _5_ }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() { return *ptr; }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() ___ { return *ptr; }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() _6_ { return *ptr; }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- * operator
- != operator

Which keyword replaces blank #6 to guarantee that the `operator*()` function will not modify the Iterator?

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() _6_ { return *ptr; }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Implementing iterators

```
class Vector {
    class Iterator {
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };

    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {

    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return ___;
    }

    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {

    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return _7_;
    }

    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Which boolean expression replaces blank #7 will evaluate to `true` when `lhs` and `rhs` "point to" the same object?

```
class Vector {
    class Iterator {

    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return _7_;
    }

    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {

    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }

    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}

        Iterator& operator++() {
            ++ptr;
            return *this;
        }

        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const ___ lhs, const ___ rhs) { }
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const _8_ lhs, const _8_ rhs) { }
```

Which type replaces blank #8 to declare the parameters to `operator!=()` when the definition is outside of the `Vector` class?

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const _8_ lhs, const _8_ rhs) { }
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) { }
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) {
    return ___;
}
```

Implementing iterators

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) {
    return _9_;
}
```

Which expression (utilizing `operator==`) replaces blank #9 to return the correct boolean for `operator!=()`?

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) {
    return _9_;
}
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- != operator

Implementing iterators

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};

bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) {
    return !(lhs == rhs);
}
```

Features to support

- ~~constructor~~
- ~~++ operator~~
- ~~* operator~~
- ~~!= operator~~

Implementing iterators

```
class Vector {
    class Iterator {
        ... // Iterator implementation
    };
    ... // Vector implementation
};
```

```
int main() {
    ...

    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    int* begin() { ~~return data;~~ }
    int* end() { ~~return data + the_size;~~ }
    ...
};
```

type mismatch

```
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    ___ begin() { ~~return data;~~ }
    ___ end() { ~~return data + the_size;~~ }
    ...
};



int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    _10_ begin() { ~~return data;~~ }
    _10_ end() { ~~return data + the_size;~~ }
    ...
};



int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Which type replaces blank #10 to match the type expected in the `for` loop?

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    _10_ begin() { ~~return data;~~ }
    _10_ end() { ~~return data + the_size;~~ }
    ...
};



int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    Iterator begin() { ~~return data;~~ }
    Iterator end() { ~~return data + the_size;~~ }
    ...
private:
    int* data;
    size_t the_size, the_capacity;
};
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    Iterator begin() { ~~return data;~~ }
    Iterator end() { ~~return data + the_size;~~ }
    ...
private:
    int* data;
    size_t the_size, the_capacity;
};
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Implementing iterators

```
class Vector {
  class Iterator {
    ...
      Iterator(int* ptr = nullptr) : ptr(ptr) {}
    ...
  };
  ...
  Iterator begin() { return ___(data); }
  Iterator end() { return ___(data + the_size); }
  ...
private:
  int* data;
  size_t the_size, the_capacity;
};
int main() {
  ...
  for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
    cout << *iter << ' ';
  }
  cout << endl;
}
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    Iterator begin() { return _11_(data); }
    Iterator end() { return _11_(data + the_size); }
    ...
private:
    int* data;
    size_t the_size, the_capacity;
};
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

Which name replaces blank #11 so that the value returned matches the return type by `begin()` and `end()`?

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    Iterator begin() { return _11_(data); }
    Iterator end() { return _11_(data + the_size); }
    ...
private:
    int* data;
    size_t the_size, the_capacity;
};
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    class Iterator {
        ...
            Iterator(int* ptr = nullptr) : ptr(ptr) {}
        ...
    };
    ...
    Iterator begin() { return Iterator(data); }
    Iterator end() { return Iterator(data + the_size); }
    ...
private:
    int* data;
    size_t the_size, the_capacity;
};
int main() {
    ...
    for (Vector::Iterator iter = vec2.begin(); iter != vec2.end(); ++iter) {
        cout << *iter << ' ';
    }
    cout << endl;
}
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
17 42 6 28
100 42 6 28
```

Implementing iterators

```
class Vector {
    class Iterator {
        ...
    };
    ...
    Iterator begin() {
        return Iterator(data);
    }
    Iterator end() {
        return Iterator(data + the_size);
    }
    ...
    private:
        int* data;
        size_t the_size, the_capacity;
};

...
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
...
100 42 6 28 17
```

```
class Iterator {
    ...
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
};


int main() {
   ...

   vec2.push_back(17);
   print_vec(vec2);

}
```

Implementing iterators

```
void print_vec() { }
```

```
int main() {
    ...

    vec2.push_back(17);
    print_vec(vec2);
}
```

Implementing iterators

```
void print_vec(const Vector& c_vec) { }
```

```
int main() {
   ...

   vec2.push_back(17);
   print_vec(vec2);
}
```

Implementing iterators

non-const

```
void print_vec(const Vector& c_vec) {

    for (Vector::Iterator iter = c_vec.begin(); iter != c_vec.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

let's add const

Iterator type

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    print_vec(vec2);
}
```

Implementing iterators

```
class Vector {
    class Iterator {
    friend bool operator==(const Iterator& lhs, const Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Iterator& lhs, const Vector::Iterator& rhs) {
    return !(lhs == rhs);
}
```

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    class ~~Iterator~~ {
    friend bool operator==(const ~~Iterator&~~ lhs, const ~~Iterator&~~ rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        ~~Iterator(~~int* ptr = nullptr) : ptr(ptr) {}
        ~~Iterator&~~ operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::~~Iterator&~~ lhs, const Vector::~~Iterator&~~ rhs) {
    return !(lhs == rhs);
}
```

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    class Const_Iterator {
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }
        int& operator*() const { return *ptr; }
    private:
         const int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Const_Iterator& lhs, const Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}
```

one more change needed...

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    class Const_Iterator {
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }
        ___ operator*() const { return *ptr; }
    private:
         const int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Const_Iterator& lhs, const Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}
```

one more change needed...

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    class Const_Iterator {
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }
        _12_ operator*() const { return *ptr; }
    private:
         const int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Const_Iterator& lhs, const Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}
```

one more change needed...

Which type replaces blank #12 to ensure that the Vector generating the Const_Iterator cannot be modified?

```
class Vector {
    ... // non-const Iterator implementation
    class Const_Iterator {
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }
        _12_ operator*() const { return *ptr; }
    private:
         const int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Const_Iterator& lhs, const Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}
```

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    class Const_Iterator {
    friend bool operator==(const Const_Iterator& lhs, const Const_Iterator& rhs) {
        return (lhs.ptr == rhs.ptr);
    }
    public:
        Const_Iterator(int* ptr = nullptr) : ptr(ptr) {}
        Const_Iterator& operator++() {
            ++ptr;
            return *this;
        }
        const int& operator*() const { return *ptr; }
    private:
         const int* ptr;
    };
    ... // Vector implementation
};
bool operator!=(const Vector::Const_Iterator& lhs, const Vector::Const_Iterator& rhs) {
    return !(lhs == rhs);
}
```

ensures Vector not modified

ensures Iterator not modified

Implementing iterators

non-const

```
void print_vec(const Vector& c_vec) {

    for (Vector::Iterator iter = c_vec.begin(); iter != c_vec.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

let's add const Iterator type

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    print_vec(vec2);
}
```

Implementing iterators

non-const

```
void print_vec(const Vector& c_vec) {

    for (Vector::Const_Iterator iter = c_vec.begin(); iter != c_vec.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    print_vec(vec2);
}
```

Implementing iterators

non-const

```
void print_vec(const Vector& c_vec) {

    for (Vector::Const_Iterator iter = c_vec.begin(); iter != c_vec.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    ~~print_vec(vec2);~~
}
```

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        ...
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) { }
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        ...
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (___ val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        ...
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (_13_ val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Which type replaces blank #13 in the ranged `for` loop?

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        ...
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (_13_ val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        Iterator begin() { return Iterator(data);}}
        Iterator end() { return Iterator(data + the_size); }
    private:
        ...
};
...
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

ranged for uses class begin() and end() methods

and automatically dereferences iterator

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
```

compilation error

Why does a compilation error result from the current implementation of `operator<<()`?

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        Iterator begin() { return Iterator(data);}}
        Iterator end() { return Iterator(data + the_size); }
    private:
        ...
};
...
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        Iterator begin() { return Iterator(data);}}
        Iterator end() { return Iterator(data + the_size); }
    private:
        ...
};
```

Need to overload begin() and end()

```
...
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        ___ begin() { return ___(data); }
        ___ end() { return ___(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        _13_ begin() { return ___(data); }
        _13_ end() { return ___(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Which return type replaces blank #13 to return a const iterator for the ranged for loop in `operator<<`()?

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        _13_ begin() { return ___(data); }
        _13_ end() { return ___(data + the_size); }
    private:
        ...
};
...
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        Const_Iterator begin() { return ___(data); }
        Const_Iterator end() { return ___(data + the_size); }
    private:
        ...
};
...
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        Const_Iterator begin() { return _14_(data); }
        Const_Iterator end() { return _14_(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Which constructor name replaces blank #14 in order to return a value of the required type for the `begin()` and `end()` methods?

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        Const_Iterator begin() { return _14_(data); }
        Const_Iterator end() { return _14_(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:
        // non-const begin() and end()
        Const_Iterator begin() { return Const_Iterator(data); }
        Const_Iterator end() { return Const_Iterator(data + the_size); }
    private:
        ...
};
...
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) {
        os << val << ' ';
    }

    return os;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:

        Iterator begin() { return Iterator(data); }
        Iterator end() { return Iterator(data + the_size); }

        Const_Iterator begin() { return Const_Iterator(data); }
        Const_Iterator end() { return Const_Iterator(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) { os << val << ' '; }
    return os;
}
```

compilation error

compilation error

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

compilation error

How can we change the `begin()` and `end()` method signatures returning a `Const_Iterator` to properly overload the methods?

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:

        Iterator begin() { return Iterator(data); }
        Iterator end() { return Iterator(data + the_size); }

        Const_Iterator begin() { return Const_Iterator(data); }
        Const_Iterator end() { return Const_Iterator(data + the_size); }
    private:
        ...
};
...

ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) { os << val << ' '; }
    return os;
}
```

```
int main() {
    ...

    vec2.push_back(17);
    cout << vec2 << endl;
}
```

compilation error

can't overload on

return type!

compilation error

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:

        Iterator begin() { return Iterator(data); }
        Iterator end() { return Iterator(data + the_size); }

        Const_Iterator begin() ___ { return Const_Iterator(data); }
        Const_Iterator end() ___ { return Const_Iterator(data + the_size); }
    private:
        ...
};
...
```

compilation error

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) { os << val << ' '; }
    return os;
}
```

compilation error

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

compilation error

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:

        Iterator begin() { return Iterator(data); }
        Iterator end() { return Iterator(data + the_size); }

        Const_Iterator begin() const { return Const_Iterator(data); }
        Const_Iterator end() const { return Const_Iterator(data + the_size); }
    private:
        ...
};
...
```

~~compilation error~~

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) { os << val << ' '; }
    return os;
}
```

~~compilation error~~

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

~~compilation error~~

Implementing iterators

```
class Vector {
    ... // non-const Iterator implementation
    ... // Const_Iterator implementation

    public:

        Iterator begin() { return Iterator(data); }
        Iterator end() { return Iterator(data + the_size); }

        Const_Iterator begin() const { return Const_Iterator(data); }
        Const_Iterator end() const { return Const_Iterator(data + the_size); }
    private:
        ...
};
...
```

```
int main() {
   ...

   vec2.push_back(17);
   cout << vec2 << endl;
}
```

ranged for automatically dereferences iterator

```
ostream& operator<<(ostream& os, const Vector& rhs) {
    for (int val : rhs) { os << val << ' '; }
    return os;
}
```

```
% g++ --std=c++11 vec_iter.cpp -o vec_iter.o
% ./vector_iter.o
...
100 42 6 28 17
```

Implementing iterators

now const version defined

~~non-const~~

```
void print_vec(const Vector& c_vec) {

    for (Vector::Const_Iterator iter = c_vec.begin(); iter != c_vec.end(); ++iter) {
        cout << *iter << ' ';
    }

    cout << endl;
}
```

~~compilation error~~

```
100 42 6 28 17
```

```
int main() {
    ...

    vec2.push_back(17);
    print_vec(vec2);
}
```

The `utility` library

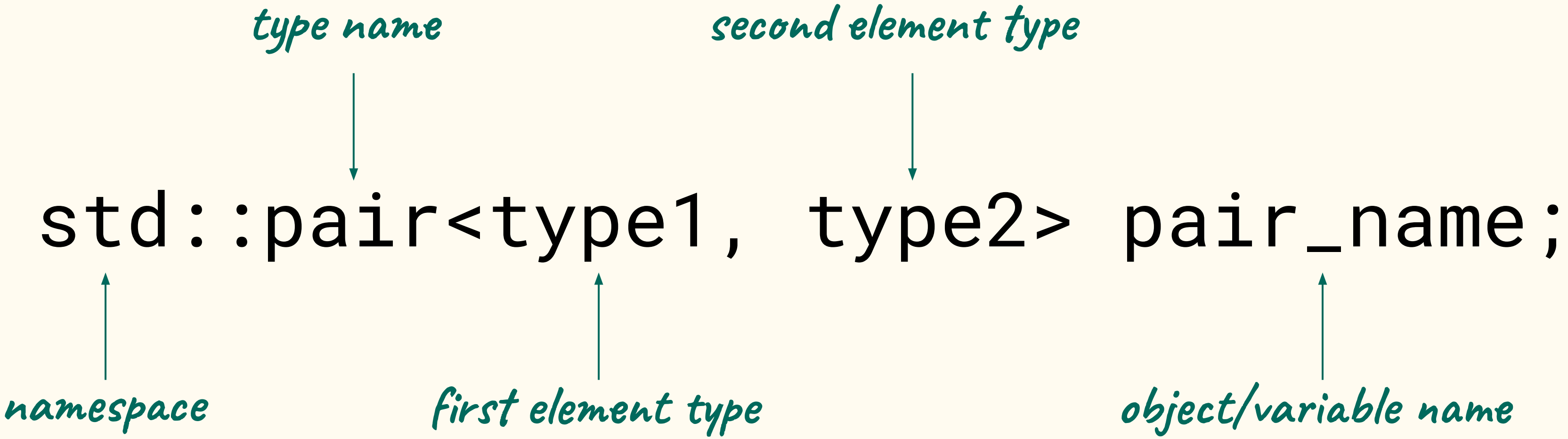
The pair type

Natural to pair data

- (x,y) coordinates
- product description/price
- student/grade
- name/ID

```
std::pair<int, int> coord;
std::pair<string, double> product;
std::pair<string, char> mark;
std::pair<string, string> employee;
```

The pair type



type name

second element type

```
std::pair<type1, type2> pair_name;
```

namespace

first element type

object/variable name

Declaring a `pair`

```
#include <utility>
using namespace std;
```

pair defined here

```
int main() {
    std::pair<int, int> coord;
    coord.first = 3;
    coord.second = 12;

    cout << '(' << coord.first << ',' << coord.second << ')' << endl;
}
```

public member variables

```
(3, 12)
```

Initializing a `pair`

```
#include <utility>
using namespace std;

int main() {
    pair<int, string> result(42, "the answer");
    cout << result.first << ": " << result.second << endl;
}
```

```
42: the answer
```

Initializing a `pair`

```
#include <utility>
using namespace std;

pair<int, string> construct_pair() {
    pair<int, string> result(42, "the answer");
    return result;
}

int main() {
```

type declaration a bit long...

```
    ~~pair<int, string>~~ result = construct_pair();
    cout << result.first << ": " << result.second << endl;

}
```

```
42: the answer
```

Initializing a pair

```
#include <utility>
using namespace std;

pair<int, string> construct_pair() {
    pair<int, string> result(42, "the answer");
    return result;
}

int main() {
```

type deduced based on assigned value

available since C++11

```
    auto result = construct_pair();
    cout << result.first << ": " << result.second << endl;

}
```

```
42: the answer
```

Initializing a `pair`

```
#include <utility>
using namespace std;

pair<int, string> construct_pair() {
    pair<int, string> result(42, "the answer");
    return result;
}
```

return type a bit long...

```
~~pair<int, string>~~ generate_pair() {
    return make_pair(42, "the answer");
}
```

defined in utility

```
int main() {
    auto result2 = generate_pair();
    cout << result2.first << ": " << result2.second << endl;
}
```

```
42: the answer
```

Initializing a `pair`

```
#include <utility>
using namespace std;

pair<int, string> construct_pair() {
    pair<int, string> result(42, "the answer");
    return result;
}
```

available since C++14

```
auto generate_pair() {
    return make_pair(42, "the answer");
}
```

return type deduced based on return value

```
int main() {
    auto result2 = generate_pair();
    cout << result2.first << ": " << result2.second << endl;
}
```

```
42: the answer
```

Restrictions on using `auto`

```
int main() {
    auto x_val = 17;

}
```

clearly an integer

Restrictions on using `auto`

```
int main() {
    auto y_val;

}
```

compilation error

Restrictions on using auto

```
void foo(auto x_val) { x_val += 1; }

int main() {

}
```

compilation error

Restrictions on using `auto`

```
auto foo(int x_val) { return x_val + 17; }

int main() {


}
```

return value clearly an int

`pair` assignment

```
#include <utility>
using namespace std;

auto generate_pair() {
    return make_pair(42, "the answer");
}

int main() {
    auto ~~result2~~ = generate_pair();
    cout << ~~result2.first~~ << ": " << ~~result2.second~~ << endl;
}
```

repeatedly writing var.first
and var.second is tedious

`pair` assignment

```
#include <utility>
using namespace std;

auto generate_pair() {
    return make_pair(42, "the answer");
}

int main() {
    auto [num, ans] = generate_pair();
    cout << ~~result2.first~~ << ": " << ~~result2.second~~ << endl;
}
```

repeatedly writing var.first and var.second is tedious

`pair` assignment

```
#include <utility>
using namespace std;

auto generate_pair() {
    return make_pair(42, "the answer");
}

int main() {
    auto [num, ans] = generate_pair();
    cout << num << ": " << ans << endl;
}
```

structured binding
(available since C++17)

```
42: the answer
```