

C++ language review



UNIVERSITY OF
ALBERTA

Phys 420/580 Lecture 2

Edit-compile-run cycle

prompt

▶ From the **UNIX terminal**:

```
$ emacs myprog.cpp &
$ g++ -o myprog myprog.cpp \
    -ansi -pedantic \
    -lm -DNDEBUG
$ ls -F
myprog.cpp  myprog*
$ ./myprog
          0      1.41421
0.0314159  1.41404
0.0628319  1.41352
0.0942478  1.41264
          |          |
        3.07876  0.0444215
        3.11018  0.0222135
```

user-created program file

```
#include <cmath>
using std::sqrt;
using std::cos;

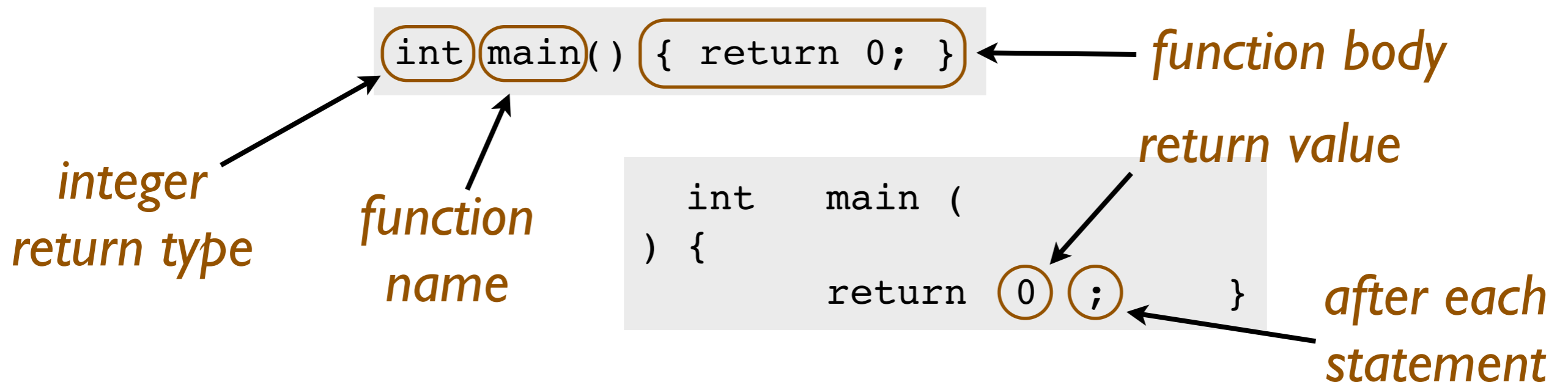
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

int main()
{
    for (int n = 0; n < 100; ++n)
    {
        const double x = M_PI*n/100.0;
        cout << setw(20) << x
              << setw(20) << sqrt(1+cos(x))
              << endl;
    }
    return 0;
}
```

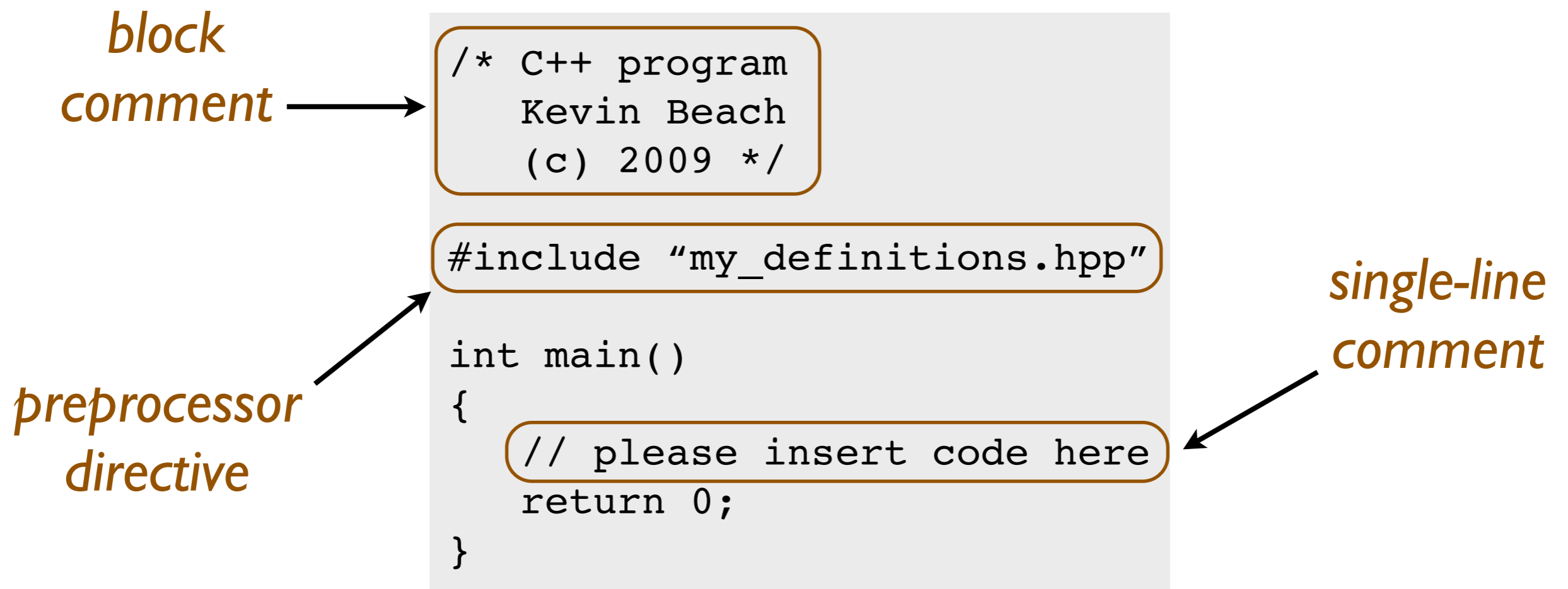
Code formatting

- ▶ C++ code is freeform:
 - ▶ code structure is indicated by semicolons and braces
 - ▶ white space has no meaning
- ▶ These are equivalent restatements of the **null program**:



Code formatting

- ▶ Single-line and block **comments** are supported
- ▶ Hash-prefixed commands are **preprocessor directives**



Properties

- ▶ The **type**, **mutability**, **scope**, and **duration** and of every **object** must be specified before it is used:
 - ▶ type and mutability are set by (prefix) keywords
 - ▶ scope and duration are controlled by where in the code an object is declared
- ▶ Every **function** acts on and returns objects of definite type

Types and their modifiers

- ▶ The integer and double-precision floating point types accept **type modifiers**
- ▶ Objects of any type may be flagged as immutable

	<i>type name</i>	<i>literal</i>	
	void		
	bool	true, false	
<i>type modifiers</i> signed, unsigned, short, long	char	'a'	<i>constant modifier</i> const
	int	-81	
	float	3.14F	
long	double	3.14	

Declarations and prototypes

- ▶ Type **declarations** and function **prototypes**

```
int sum(int x, int y) { return x+y; }
int prod(int, int);

const int A = 5;

int main()
{
    int a;
    int b = 6;
    a = sum(b,A); // give a the value 11
    return 0;
}
```

*function
prototype*

*integer
variable
declaration*

Declarations and prototypes

- ▶ Objects survive until their current code block terminates
- ▶ A variable name may be temporarily obscured

global variable

```
int k;  
  
int main()  
{  
    int i = 0;  
    {  
        int i, j ;  
        i = j = 1;  
    }  
    k = i; // k is 0 here  
    return 0;  
}
```

Algebraic operators

modify in place

```
int a = (5+7*3)/2; // a is 13
a = a - 2; // 11
a += 4; // 15
a /= 3; // 5
int b = a%4; // b is 1
int c = ++a/2; // a is 6, c is 3
int d = a--/3; // a is 5, d is 2
```

*pre- and postfix
"side-effects"*

```
#include <cmath>
using std::pow; using std::sqrt;
```

*functions
from the
math library*

```
double x0 = 2.0*2.0;
double x1 = pow(2.0,2.5);
double x2 = x0*2.0;
double eps = sqrt(x0*x2)-x1;
```

Boolean (logical) operators

assignment operator

```
#include <cassert>
```

```
int a = 5;  
int b = 7;
```

test for equality

```
assert(a != b);  
assert(!( a == b ));
```

run-time checks

```
bool test1 = a < 2*b+5 and a != b;  
bool test2 = a*b >= 3 or a*b <= -3;  
bool test3 = b > a > 3;
```

legal but misleading

```
assert(test1);  
assert(test2);  
assert(!test3);
```

Bitwise operators

*enumerated
type*

```
enum directions { N = 1, E = 2, S = 4, W = 8 };  
const uint8_t opt1 = 020; // 2*8 == 16  
const uint8_t opt2 = 0x20; // 2*16 == 32
```

```
unsigned char flags = N | W;  
assert( (flags & N) and (flags & W) );
```

octal and hex

set,

clear,

and toggle

bits

```
flags |= S | E;  
assert( flags == N | S | E | W );
```

test bits

```
flags &= ~S;  
assert( flags == N | E | W );
```

```
flags ^= N | E | opt1;  
assert( flags == W | opt1 );  
flags ^= opt1 | opt2;  
assert( !(flags & opt1) and (flags & opt2) );
```

Control structures

- ▶ C++ provides standard **looping** and **branching** constructs:

```
int x = 1, y = 1;  
int n = 5;  
while (n > 0) { x *= 2; --n; }  
do { y *= 2; } while (y < 32);  
assert(x == y and y == 32);
```

```
const int M = (x < 0 ? -x : x);
```

```
int div5 = 0;  
for (int m = 0; m < M; ++m)  
    if (m%5 == 0)  
        cout << ++div5 << endl;  
    else  
        do_something();  
assert(div5 == 7);
```

looping

branching

Function arguments

- ▶ Functions **arguments** are **passed by value**, which prevents side effects
- ▶ Changes can be made to propagate outside the function by passing a **reference** instead

```
int thrice(int x) { return 3*x; }  
void triple(int &x) { x *= 3; }
```

```
int x = 3;  
const int y = thrice(x); // y is 3  
triple(x); // x is now 9
```

*referencing
operator*

Command-line arguments

```
#include <cstdlib>
using std::atoi; using std::atof;
using std::exit;
#include <iostream>
using std::cerr; using std::cout;
using std::endl;

int main(int argc, char* argv[])
{
    if (argc != 3)
    { cerr << "Two arguments required" << endl; exit(1); }

    int i = atoi(argv[1]); // argv[0] is "myprog"
    double x = atof(argv[2]);

    cout << "Int: " << i << " FP: " << x << endl;
    return 0;
}
```

```
$ emacs myprog.cpp &
$ g++ -o myprog myprog.cpp \
      -ansi -pedantic \
      -lm -DNDEBUG
$ ./myprog
Two arguments required
$ ./myprog 5 -3.88
Int: 5 FP: -3.88
```

Writing to the terminal

```
#include <iostream>
using std::cout;
using std::endl;

#include <iomanip>
using std::setw;

#include <cmath>

main()
{
    cout.setf(std::ios::scientific);
    cout.precision(8);
    cout << setw(16) << M_PI << endl;
    return 0;
}
```

Composite objects

- ▶ **arrays** are groups of identically typed objects stored contiguously in memory
- ▶ **structures** and **classes** bundle objects of arbitrary type

```
bool active[3];  
active[0] = active[1] = active[2] = true;  
  
struct particle { double m, x, vx; };  
particle gas[100]; particle test;  
  
gas[0].m = 1.0;  
test.vx = -5.5;
```

*member
operator*

*zero-based
array
indexing*

Composite objects

- ▶ Classes may also have **methods** associated with them
- ▶ Methods are functions belonging to a class that act on its internal data

constructor / initialization list

```
class particle
{
public:
    double m, x, vx;
    particle(double m_, double x_, double vx_) :
        m(m_), x(x_), vx(vx_) {}
    double energy(void) { return 0.5*vx*vx/m; }
};
```

```
particle p(2.0,0.0,3.0);
const double E = p.energy();
```

method

Passing composite objects

- ▶ Arrays are passed as **pointers** to the first element
- ▶ Structures and classes should be passed by reference
- ▶ Only a **memory address** rather than the data is copied

```
double dot_product(double u[], double v[], int N);  
double dot_product(double* u, double* v, int N);
```

pointer dereferencing

```
double momentum(const particle &p)  
{ return p.m*p.vx; }
```

*reference to a
constant object*

```
void evolve_no_accel(particle &p, double dt)  
{ p.x += p.vx*dt; }
```

Passing composite objects

- ▶ arrays don't know their own size and are thus dangerous

```
#include <cassert>
double sum_squares(const double x[], int N)
{
    if (N == 1) return x[0]*x[0];
    else if (N > 1)
    {
        double sum = x[0]*x[0];
        for (int i = 1; i < N; ++i)
            sum += x[i]*x[i];
        return sum;
    }
    else assert(false);
}
const double v[3] = { 1.0, 2.0, -3.0 };
double norm_v = sum_squares(v, 5);
```

*forces a
run-time
error*

*memory error:
"segmentation
fault"*

Templates

- ▶ **Templates** provide a mechanism for adding compile-time pattern matching to classes and functions:

```
template <typename T>
T abs_value(T x)
{
    if (x < 0) return -x;
    else return x;
}

template <int y>
void increment_by(int &x) { x += y; }

int a = -5;
int b = abs_value(a); // b is 5
increment_by<3>(b); // 8
```

STL container classes

- ▶ The C++ standard library provides a variety of **dynamically-allocated** data structures:

```
#include <vector>
using std::vector;

double sum_squares(const vector<double> &v)
{
    assert(v.size()>0);
    double sum = 0.0;
    for (int i = 0; i < v.size(); ++i)
        sum += x[i]*x[i];
    return sum;
}

vector<double> u; double u0 = 0.5;
while(u.size() < 10) u.push_back(u0*=2.0);
const double N = sum_squares(u);
```

STL container classes

- ▶ **sequence containers** →

- ▶ **$O(1)$ element access**
- ▶ **$O(N)$ insertion**

```
#include <vector>
using std::vector

#include <deque>
using std::deque
```

- ▶ **associative containers** →

- ▶ **$O(\log N)$ lookup**
- ▶ **$O(1)$ insertion**

```
#include <set>
using std::set

#include <map>
using std::map
```

`list, slist, multiset, multimap, stack, queue, ...`

Pointers and iterators

- ▶ **Pointers** point to data at a particular location in memory
- ▶ **Iterators** are pointer-like abstractions that are provided by the C++ container classes

```
double a[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
vector<double> v(a,a+5);

for (int i = 0; i < v.size(); ++i)
    assert(a[i] == v[i]);

for (double* step = a; a < a+5; ++step)
    do_work_on(*step);

for (vector<double>::iterator step = v.begin();
     step != v.end(); ++step)
    do_work_on(*step);
```

Generic programming

```
#include <vector>
using std::vector;

template <class Iter>
double sum_squares(Iter begin, Iter end)
{
    assert(begin != end);
    Iter p = begin;
    double sum = (*p)*(*p);
    while (++p < end) sum += (*p)*(*p);
    return sum;
}

const double A[] = { 2.7, -5.5, 100.1 };
const int B[] = { 1, 2, 3, 4, 5 };
vector<double> v(A,A+3);

sum_squares(v.begin(),v.end());
sum_squares(B,B+5);
```

File output and input

```
#include <iostream>
using std::endl;
#include <fstream>
using std::ofstream; using std::ifstream;
#include <cassert>

main()
{
    ofstream fout("myfile.txt");
    fout << "1 2 3 4 5" << endl;
    fout << "6 7 8 9 10" << endl;
    fout.close();
    ifstream fin("myfile.txt");
    vector<int> v;
    while(fin) { int i; fin >> i;
                v.push_back(i); }
    fin.close();
    assert(v.size() == 10);
    return 0;
}
```

Our overall approach

- ▶ Programming style will be more procedural than OO:
 - ▶ Treat C++ as a syntactically cleaner version of C
 - ▶ Make use of templates, generic programming, and STL data structures
- ▶ Things we'll largely ignore:
 - ▶ class inheritance and polymorphism
 - ▶ virtual functions
 - ▶ exception handling