COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Today

• Templates

• Operator Overloading
Templates

- Syntactically similar to Java generics
- Used in similar ways (containers!)
- But implemented very differently!

⇒ Noticeable differences for developers!
max for double

double max(unsigned int num, double * arr)
{
    double d = arr[0];
    for (unsigned int i=1;i<num;i++)
        if (d < arr[i]) d = arr[i];
    return d;
}
max for int

int max(unsigned int num, int * arr)
{
    int d = arr[0];
    for (unsigned int i=1; i<num; i++)
        if (d < arr[i]) d = arr[i];
    return d;
}
max for float

float max(unsigned int num, float * arr)
{
    float d = arr[0];
    for (unsigned int i=1;i<num;i++)
        if (d < arr[i]) d = arr[i];
    return d;
}
#define make_max_func(type) \
    type max_##type(int num, type * arr) \
{
    \
    unsigned int i;
    \
    type d = arr[0];
    \
    for (i=1;i<num;i++)
    \
        if (d < arr[i]) d = arr[i];
    \
    return d;
    
}
#define make_max_func(type) \
  type max(unsigned int num, type * arr) \{
     type d = arr[0]; \ 
     for (unsigned int i=1;i<num;i++) \ 
       if (d < arr[i]) d = arr[i]; \ 
     return d; \ 
  }
max with Templates in C++

template <class T>
T max(unsigned int num, T * arr)
{
    T d = arr[0];
    for (unsigned int i=1;i<num;i++)
        if (d < arr[i]) d = arr[i];
    return d;
}
Templates

Templates are like Macros:

- Templates must be defined in headers
- The compiler generates no code for the template

⇒ Code is only generated if the template is used
⇒ Many compile errors are generated (only) when the template is instantiated
Templates

Templates are not like Macros:

• An expanded template is not (necessarily) inlined into the code

⇒ Overhead of a function call may apply!

⇒ Savings in code size may apply!

• You can use overloading with templates, but not default values for arguments
Templates

- Unlike Macros, the parameters in Templates are always types

- `class T` does not imply that `T` must be a C++ class, primitive types work as well!

- `typename T` also works in ISO-C++\(^1\)

Templates and structs

template <typename T> struct slist
{
    slist<T> * next;
    T value;
}

template <typename T> class stack {
    private:
        slist<T> * top;
    public:
        stack();
        void push(T);
        void pop();
        T top();
        ~stack();
};
Providing the Implementation

template <class T> T stack<T>::top()
{
    if (top == NULL)
        throw "stack empty";
    return top->value;
}
Nested Template Instantiation

The following is problematic for the C++ compiler:

stack<stack<int>> s;

Why?
Solution

The lexer is happy with:

```cpp
stack<stack<int> > s;
```

Java’s lexer works for both variants. Compiler writers are still getting better...
Operator Overloading in C++

You can not:

- define new Operators (such as + > or >>)
- overload "sizeof", ".", ".*", "::" and "?:"
- change the precedence of operators
- change the arity of operators
- change the operation of operators on primitive types
Overloading Operations

- As simple as defining a function with appropriate argument types and return type
- One of the types must be a class (some compilers also allow enumerations)
Example

Vector operator * (const Matrix&, const Vector&) 
operator is a keyword. const is not required (but makes sense in this context).
Calling Overloaded Operators

Vector v;
Matrix m;
Vector r1 = m * v;
Vector r2 = m.operator + (v);
Defining Operators (globally)

\[ x \oplus y \Rightarrow \text{operator } \oplus (x, y) \]
\[ \oplus x \Rightarrow \text{operator } \oplus (x) \]
\[ x \oplus \Rightarrow \text{operator } \oplus (x, 0) \]

The last operator only applies for \( \oplus \in \{++, --\} \).
class Complex {
    Complex operator +(Complex);
    Complex operator -(Complex);
    Complex operator *(Complex);
    Complex operator /(Complex);
}

Another Example
Defining Operators as Member Functions

\[ x \oplus y \Rightarrow x.\text{operator} \oplus (y) \]
\[ \oplus x \Rightarrow x.\text{operator} \oplus () \]
\[ \oplus x \Rightarrow x.\text{operator} \oplus (0) \]
\[ x(y,z,\ldots) \Rightarrow x.\text{operator} () (x, y, \ldots) \]
\[ x[y] \Rightarrow x.\text{operator} [] (y) \]
\[ x->m \Rightarrow (x.\text{operator}())->m \]

The last three operators can only be overloaded using member functions.
Example: Unary Plus

class Complex {
    const Complex operator +() const;  
}

const Complex& Complex::operator+() const
{
    return *this;
}

const is necessary to prevent something like "+v1 = v2".
Example: Binary Plus

class Complex {
    const Complex operator +(const Complex&) const;
}
const Complex& Complex::operator+
    (const Complex &o) const {
    return Complex(rv + o.rv, iv + o.iv);
}

const is necessary to prevent something like “+v1 = v2”.
Best Practice Rules

• Do **not** allocate the return value of an operator with `new`, the caller will most likely forget to do a `delete`

• `r = r + b` should be the same as `r += b`; trick:

```cpp
T operator +(T &r1, T&r2) {
    T temp = r1;
    return temp += r2;
}
```
Example: “()”

class Polynom {
    double operator()(double x) const;
}

double Polynom::operator()(double x) const {
    double d = coeff[deg];
    for (int i=deg-1;i>=0;i--) d = d*x + coeff[i];
    return d;
}
Example: “()”

With the above definitions, one could then do:

```
Polynom p;

p(4.2);
```

to evaluate \( p \) at 4.2.
Example: “["[]"]”

template <typename T>
class Vector {
    T &operator[](int i);
}
template <typename T>
T& Vector::operator[](int i)
{
    if ( (i<0) || (i>= len) ) error();
    return v[i];
}


Questions
Question

Given an overloaded operator “[]” that returns a row (of type Vector) for a Matrix, what will the following code do?

Matrix m;

m[3,4];
Example: Output

The C++ equivalent of Java’s toString methods is:

```cpp
ostream & operator << (ostream &, const T&);
```

The first argument should be returned to allow

```cpp
cout << a << b << endl;
```

which should be read as

```cpp
((cout << a) << b) << endl;
```
Example: Input

For input, the class needs to overload “\(>>\)”:  
\[\text{istream } & \text{ operator } >> (\text{istream } &, \ T&);\]

The first argument should be returned to allow  
\[\text{cint } >> a \text{ } >> \text{ b;}\]

which should be read as  
\[((\text{cin } >> a) \text{ } >> \text{ b}) \text{ } >> \text{ endl;}\]
Questions?