

POLYMORPHIC VS. OVERLOADED FUNCTIONS

- **Problem.** We want to have a function `max_array` that cycles through an array with $n > 0$ elements and returns the maximum element found in that array. Something like this:

```
int max_array (int* a, int n) {
    int ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

- Wait, we did not say we want a function on integers only!
 - Let's be more precise: we need a function that accepts as argument an array of any type that can be compared using the normal relational operators.
- We could overload the function `max_array`, but this does not take care of the word “any” above.
- We have **overloaded** functions in C++, but now we want **truly polymorphic functions**.
- Can we do it with a macro?

THE HARD WAY

- We define a macro:

```
#define define_max(type) type max_array (type* a, int n) { \
    type ret = a[0]; \
    for (int i = 1; i < n; i++) \
        ret = a[i] > ret ? a[i] : ret; \
    return ret; \
}
```

- Say we want to use `max_array` on arrays of `int`, `float`, and `char`. We then do:

```
define_max(int);
define_max(float);
define_max(char);
```

THE HARD WAY (CONT'D)

Source code:

```
#define define_max(type)      \
    type max_array (type* a, int n) { \
        type ret = a[0]; \
        for (int i = 1; i < n; i++) \
            ret = a[i] > ret ? a[i] : ret; \
        return ret; \
    }
```

```
define_max(int);
```

```
define_max(float);
```

```
define_max(char);
```

```
int main () {
    float af[] = {1.0, 2.2, 0.5};
    int ai[] = {1, 2, 0};
    char ac[] = {'a', 'd', 'b'};
    int mi = max_array(ai,3);
    float mf = max_array(af,3);
    char mc = max_array(ac,3);
}
```

Generated code: (a mere overloaded function)

```
int max_array (int* a, int n) {
    int ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

```
float max_array (float* a, int n) {
    float ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

```
char max_array (char* a, int n) {
    char ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

PROBLEMS

- Suitable for small functions, more problems as the function size grows
 - ... including those pesky backslashes at the end of each line in the macro definition...
- We have to insert a call to `define_max` in our source code for each instance of the function `max_array`.
 - Still not a true polymorphism.

TEMPLATES

- We can in fact define a **generic function**:

```
template <class Object> Object max_array (Object* a, int n) {  
    Object ret = a[0];  
    for (int i = 1; i < n; i++)  
        ret = a[i] > ret ? a[i] : ret;  
    return ret;  
}
```

- This is not a function definition, but a **template** which can be turned into a real function as needed.
 - * Closer in fact to a macro than to a function definition.
 - C++ then uses this template to automatically generate the specific **instance** of the function.
- Advantages over macros:
 - Easier to write. You write templates in the same way you write functions.
 - Instances are generated automatically (no call to `define_max` necessary).

TEMPLATES (CONT'D)

Source code:

```
template <class Object>
Object max_array (Object* a, int n) {
    Object ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

```
int main () {
    float af[] = {1.0, 2.2, 0.5};
    int ai[] = {1, 2, 0};
    char ac[] = {'a', 'd', 'b'};
    int mi = max_array(ai,3);
    float mf = max_array(af,3);
    char mc = max_array(ac,3);
}
```

Generated code:

(a mere overloaded function)

```
int max_array (int* a, int n) {
    int ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

```
float max_array (float* a, int n) {
    float ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

```
char max_array (char* a, int n) {
    char ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}
```

SPECIALIZATION

- Problems remaining to be solved: strings (i.e., `char*`) and the like.
 - They are also comparable, but with `strcmp` instead of the normal relational operators.
 - So the following piece of code will not work as expected:

```
char* c[] = {"alpha", "gamma", "beta"};
cout << max_array(c,3) << "\n";
```

- It would be nonetheless nice to be able to use our template on strings too, since after all they are still comparable (it's just the syntax that is different).

SPECIALIZATION

- Problems remaining to be solved: strings (i.e., `char*`) and the like.
 - They are also comparable, but with `strcmp` instead of the normal relational operators.
 - So the following piece of code will not work as expected:

```
char* c[] = {"alpha", "gamma", "beta"};
cout << max_array(c,3) << "\n";
```

- It would be nonetheless nice to be able to use our template on strings too, since after all they are still comparable (it's just the syntax that is different).
 - We actually can: we **specialize** out template for the particular case of strings.

```
template <class Object>
Object max_array (Object* a, int n) {
    Object ret = a[0];
    for (int i = 1; i < n; i++)
        ret = a[i] > ret ? a[i] : ret;
    return ret;
}

// Specialization (no template):
char* max_array (char** a, int n) {
    char* ret = a[0];
    for (int i = 1; i < n; i++)
        if (strcmp(a[i],ret) > 0)
            ret = a[i];
    return ret;
}
```

CLASS TEMPLATES

- Same idea (and same features) as for functions...
 - ...only with a very baroque syntax.
- Declarations:

```
template <class content_t> class list {
    cons_cell<content_t>* clone_cons (cons_cell<content_t>*) const;
protected:
    cons_cell<content_t>* content;
public:
    list(void);
    list(const list&);
    list(cons_cell<content_t>*);
    ~list(void);
    const list& operator=(const list&);
    int null(void) const;
    content_t car(void) const;
    void cdr(void);
    void cons(content_t);
    void rmth(int = 0);
    void print(void) const;
};
```

```
template <class content_t>
struct cons_cell {
    content_t car;
    cons_cell<content_t>* cdr;
    cons_cell(content_t ,
              cons_cell<content_t>* = 0);
};

template <class content_t>
ostream& operator<<
    (ostream&,
     const list<content_t>&);
```

CLASS TEMPLATES (CONT'D)

- Definitions:
 - You have to **repeat** the template declaration in each and every type used in the implementation.
 - As well, **declarations and definitions have to reside in the same file.**
 - Definitions are in effect a form of declaration (you do not in fact define member functions, just **templates** for member functions).

```
template <class content_t>
cons_cell<content_t>::cons_cell (content_t val, cons_cell<content_t>* rest) {
    car = val; cdr = rest;    }
```

```
template <class content_t> list<content_t>::list (void) { content = 0; }
```

```
template <class content_t> list<content_t>::list (cons_cell<content_t>* c)
: content (c) { }
```

```
template <class content_t> list<content_t>::list(const list<content_t>& l) {
    content = clone_cons(l.content);    }
```

```
template <class content_t> list<content_t>::~~list (void) {
    while (content != 0) cdr();    }
```

CLASS TEMPLATES (CONT'D)

```
template <class content_t>
const list<content_t>& list<content_t>::operator=(const list<content_t>& rhs) {
    if (this != &rhs) content = clone_cons(rhs.content);
    return *this;    }

template <class content_t>
cons_cell<content_t>* list<content_t>::clone_cons (cons_cell<content_t>* c) const {
    if (c == 0) return 0;
    return (new cons_cell<content_t>(c -> car, clone_cons(c -> cdr))); }

template <class content_t> int list<content_t>::null (void) const {
    return content == 0;    }

template <class content_t> content_t list<content_t>::car (void) const {
    return content -> car;    }

template <class content_t> void list<content_t>::cdr (void) {
    if (content != 0) { cons_cell<content_t>* tmp = content;
        content = content -> cdr; delete tmp; }
}

template <class content_t> void list<content_t>::cons(content_t c) {
    content = new cons_cell<content_t>(c,content);
}
```

CLASS TEMPLATES (CONT'D)

```
template <class content_t> void list<content_t>::rmth (int which) {
    cons_cell<content_t>* place = content;
    for (int i = 0; i < which - 1; i++) {
        if (place == 0) return;
        place = place -> cdr;    }
    if (place !=0 && place -> cdr != 0) {
        cons_cell<content_t>* to_delete = place -> cdr;
        place -> cdr = place -> cdr -> cdr; delete to_delete;
    }
}
```

```
template <class content_t> void list<content_t>::print(void) const {
    cons_cell<content_t>* iter = content;
    cout << "(";
    while (iter != 0) {
        cout << iter -> car; iter = iter -> cdr;
        if (iter != 0) cout << ",";    }
    cout << ")";
}
```

```
template <class content_t>
ostream& operator<< (ostream& out, const list<content_t>& value) {
    value.print(); return out; }
```

IT'S A TYPE, IT'S A PLANE

- Consider the following code:

```
int y;  
template <class T> void g(T& v) {  
    T::x(y);  
}
```

- The statement `T::x(y)` can be

IT'S A TYPE, IT'S A PLANE

- Consider the following code:

```
int y;  
template <class T> void g(T& v) {  
    T::x(y);  
}
```

- The statement `T::x(y)` can be
 - * the function call (member function `x` of `T` applied to `y`), or
 - * the declaration of `y` as a variable of type `T::x`.

IT'S A TYPE, IT'S A PLANE

- Consider the following code:

```
int y;  
template <class T> void g(T& v) {  
    T::x(y);  
}
```

- The statement `T::x(y)` can be
 - * the function call (member function `x` of `T` applied to `y`), or
 - * the declaration of `y` as a variable of type `T::x`.
- Resolution: unless otherwise stated, an identifier is assumed to refer to something that is **not** a type or template.
 - * If we want something else, we use the keyword `typename`:

```
T::x(y);           // function x of T applied to y  
typename T::x(y); // y is a variable of type T::x
```

A CONCRETE EXAMPLE

- The C++ compiler can get confused easily about what is a type and what is not **especially when classes are defined inside template classes:**

```
template <class content_t>
class list {
    ...
public:
    ...
    class iterator {
        ...
    };
    iterator begin(void);
    iterator end(void);
};
...
template <class content_t>
list<content_t>::iterator list<content_t>::end(void) { // line 222
    list<content_t>::iterator ret(0);
    return ret;
}
```

```
list.h:222: error: expected constructor, destructor, or type conversion
before "list"
```

A CONCRETE EXAMPLE (CONT'D)

- We **have** a type in the noted place, so we conclude that the compiler gets confused. We then help it:

```
typename list<content_t>::iterator list<content_t>::end(void) {  
    list<content_t>::iterator ret(0);    // line 223  
    return ret;  
}
```

```
list.h:223: error: dependent-name `list<content_t>::iterator' is parsed  
as a non-type, but instantiation yields a type
```

- One more confusion, one more hint:

```
typename list<content_t>::iterator list<content_t>::end(void) {  
    typename list<content_t>::iterator ret(0);  
    return ret;  
}
```

- The keyword `typename` is a hint for the compiler, which at times must be used.