Chapter 24
Templates

Template

A template is a way of writing a generic procedure or class.

Templates look good, but there are no compilers which implement the standard. (As of 2003)

In short, templates will be a good thing when they grow up.

Templates the hard way

Definition phase

```
#define make_max(type) type max(type d1, type d2) { \
        if (d1 > d2)                                \
            return (d1);                            \
        return (d2);                                \
    }
```

Generation phase:

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

Usage phase:

```
int main(void) {
    float f = max(3.5, 8.7);
    int   i = max(100, 800);
    char ch = max('A', 'Q');
```

What's generated

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

```
max(int d1, int d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

```
max(float d1, float d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

```
max(char d1, char d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

```
int main(void) {
  float f = max(3.5, 8.7);
  int   i = max(100, 800);
  char ch = max('A', 'Q');
```

Templates the easy way

Definition phase:

```
template<class kind>
kind max(kind d1, kind d2) {
    if (d1 > d2)
        return (d1);
    return (d2);
}
```

Generation phase

Automatic

Usage phase:

```
int main(void) {
    float f = max(3.5, 8.7);
    int   i = max(100, 800);
    char ch = max('A', 'Q');
    int  i2 = max(600, 200);
```

What's generated

```
max(int d1, int d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

auto generated

```
max(float d1, float d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

auto generated

```
int main(void) {
  float f = max(3.5, 8.7);
  int   i = max(100, 800);
  char ch = max('A', 'Q');
```

```
max(char d1, char d2) {
   if (d1 > d2)
       return (d1);
   return (d2);
}
```

auto generated

Function Specialization

This won't work (at least it won't do what we expect.)

```
char *name1 = "Able";
char *name2 = "Baker";

std::cout << max(name1, name2) << '\n';
```

A specialized version

```
char *max(char *d1, char *d2) {
    if (strcmp(d1, d2) < 0)
        return (d1);
    return (d2);
}
```

Template Example

```
#include <iostream>
#include <string.h>

// A template for the "max" function

template<class kind>
kind max(kind d1, kind d2) {
    if (d1 > d2)
        return (d1);
    return (d2);
}

// A specialization for the "max" function
//   because we handle char * a little differently
char *max(char *d1, char *d2) {
    if (strcmp(d1, d2) > 0)
        return (d1);
    return (d2);
}
```

Class Templates

```
#include <stdlib.h>
#include <iostream>


const int STACK_SIZE = 100;     // Maximum size of a stack


/*****************************************************
 * Stack class                                   *
 *                                              *
 * Member functions                              *
 *     stack -- initalize the stack.             *
 *     push -- put an item on the stack.          *
 *     pop -- remove an item from the stack.      *
 *****************************************************/
// The stack itself
template<class kind>
class stack {
   private:
      int count;             // Num. of items in the stack
      kind data[STACK_SIZE];  // The items themselves
```

Class Templates

```
public:
    // Initialize the stack
    stack(void) {
        count = 0;  // Zero the stack
    }

    // Push an item on the stack
    void push(const kind item) {
        data[count] = item;
        ++count;
    }

    // Pop an item from the stack
    kind pop(void) {
        // Stack goes down by one
        --count;
        // Then we return the top value
        return (data[count]);
    }
};
```

Member functions

```
/********************************************************
 * stack::push -- push an item on the stack.           *
 *                                                      *
 * Warning: We do not check for overflow.               *
 *                                                      *
 * Parameters                                           *
 *      item -- item to put in the stack                *
 ********************************************************/
template<class kind>
inline void stack<kind>::push(const kind item)
{
   data[count] = item;
   ++count;
}
```

Class Specialization

```
inline void stack<char *>::push(const char * item)
{
   data[count] = strdup(item);
   ++count;
}
```

Implementation Difficulties

iinteger.cpp defines

```
integer& operator *(const integer& i1,
                    const integer& i2);
```

square.cpp defines

```
template<typename item>item square(
     const item &i) {
           return (i*i);
 }
```

main.cpp defines

```
integer i1, i2;
i1 = 5;
i2 = square(i1);
```

The problem

main.cpp needs to generate code for `sum<integer>`. It knows how to multiple integers, but does not know what the body of `sum` looks like.

sum.cpp knows what the body of `sum` looks like, but does not know how to multiple the `integer` type. (It also does not know that `sum<integer>` is needed.)

The official solution

The **export** keyword.

```
export template<typename item>
item square(const item& i) {
   return (i*i);
}
```

Denotes a template that may be used in another module. The file containing the export template must be compiled before the file using it.

Problem: No one implements the standard.

Unofficial Solutions

1) Make all template bodies inline and put them in headers.

2) Require that the code for all the types a template can use be included in the template definition file:

```
#include "integer.h"
template<typename item>item square(...);

// Force generation of the code
square<integer>(const integer &i);
```