

Introduction to C++

Friends, Nesting, Static Members, and Templates

Topic #7

Relationship of Objects

- Friends, Nesting
- Static Members

Template Functions and Classes

- Reusing Code
- Template Specializations
- Implicit and Explicit Instantiations
- Partial Specializations
- Discuss Efficiency Implications

Friends

- Remember that with data hiding and encapsulation, we force clients to manipulate private data through public member functions. By declaring friends, we allow non-member functions or member functions of other classes access to private data.
- A class can declare another class to be a friend.
- When we do this, the first class gives all member functions of the second class permission to access all of its protected and private information (not the other way around; a class cannot declare itself a friend of another class).

Friend Classes

- By declaring an entire class as a friend, all members of the friend class have permission to access the protected and private members of the declaring class. This grants special privileges to the friend class' member functions. However, declaring a class to be a friend does not grant any privileges to functions that may be called by member functions of the friend class.

```
class declaring_class {  
    friend class class_name1, //class_name1 is a friend  
        class_name2; //class_name2 is a friend  
    ...  
};
```

Friend Classes

- A class can be declared a friend before the friend class is defined. This is because the friend declaration only needs an incomplete declaration of the friend class. An incomplete declaration informs the compiler that the friend class may be defined later.

```
class declaring_class {  
    //class_name is not previously declared or defined, so  
    //an incomplete declaration is used  
    friend class class_name; //class_name is a friend class  
    ...  
};
```

Friend Classes

- **When a class (declaring class) declares another class (friend class) to be a friend, the friend class' member functions can invoke the protected or private member functions or use the protected or private data members of the declaring class.**
- **To do so, the friend class must have access to an object of the declaring class' type.**
- **This can be done by having an object of the friend class as a data member or a pointer to an object of the friend class.**

Non-Member Friend Functions

- A non-member function can be declared a friend by placing the following statement in the declaring class. The declaration of a friend function takes the form of a function prototype statement, preceded by the keyword friend.
- We saw this type of friend with operator overloading.
- Typically, friend functions are designed with formal arguments, where clients pass either an object, the address of an object, or a reference to an object as an argument.

```
class declaring_class_name {  
    friend return_type function_name(arg_list);  
};
```

Member Function Friends

- We can declare a member function to be a friend by placing the following statement in the declaring class.
- The declaration of a friend member function takes the form of a member function prototype statement, preceded by the keyword `friend`.
- Member functions are declared using their class name followed by the scope resolution operator.
- The friend member function must have an object of the class to which it is a friend -- from a formal argument, as a local object in the member function's implementation, as a data member in the member function's class, or as a global object.

```
class declaring_class_name {  
    friend return_type class_name::function_name(arg_list);  
};
```


Member Function Friends

- **But... A member function cannot be declared a friend before it is first declared as a member of its own class.**
- **Unlike a friend class, where we can have an incomplete declaration, friend member function declarations (in our declaring class) cannot take place until after the member functions have been declared (in their own class).**
- **If two or more classes share the same friend class or function, those classes share a mutual friend. This means that more than one class gives a single friend class or function permission to access their members.**

Nesting

- Nesting is quite different than declaring friends. Friends provide special privileges to members; whereas, nesting restricts the scope of a class' name and can reduce the global namespace pollution.
- A class' definition can be placed inside the public, protected, or private sections of another class. The purpose of this is to hide the nested class' name inside another class, restricting access to that name.
- It does not give either the outer class or the nested class any special privileges to access protected or private members of either class.
- A nested class does not allow the outer class access to its hidden members....the name of the nested class is hidden.

Nesting

- If a class is defined within another class in the public section, the nested class is available for use the same as objects defined for the outer class.
- To define objects of a public nested class, the name of the nested class must be qualified by the outer class' name (i.e., `outer_class::nested_class`).

```
class outer_class {  
    public:  
        class nested_class {  
            public:  
                nested_class(); //nested class' constructor  
            ...  
        };  
};
```

- `outer_class::nested_class object; //define an object`

Nesting

- If a class is defined within another class in the private section, the nested class is available for use only by the member functions of the outer class and by friends of that class. Clients cannot create objects of the nested class type.
- In the implementation of a nested class' member functions, where the interface is separate from the implementation, an additional class name and scope resolution operator is required.

```
//nested class' constructor implementation
outer_class::nested_class::nested_class() {
    ...
}
```

Nesting Example

```
#include "employee.h"
class tree { //binary tree class
public:
    class node; //forward reference to node
    tree(); //default constructor
    tree(const tree &); //copy constructor
    ~tree(); //destructor
    tree & operator = (const tree &); //assignment op
    void insert(const employee &); //insert employee
    void write() const; //write employee info
private:
    node* root; //root node of tree
    friend static void copy(node* &, const node*);
    friend static void destroy(node*);
    friend static tree::node* add(node*, node*);
};
```

- Notice that the friend declarations must occur after node is declared to be a private class of tree. Also notice that the definition must still qualify node as tree::node.

Tree Implementation File

```
#include "tree.h"
class tree::node { //node for binary tree
public:
    node() :           //default constructor
        left(0),
        right(0) {
    }
    node(const employee &obj) : //one arg constructor
        emp(obj),
        left(0),
        right(0) {
    }
    employee emp;           //employee object
    node* left;            //left child node
    node* right;           //right child node
};
```

Tree Implementation File

```
static void copy(tree::node* &new_node,
                const tree::node* old_node) {
    if(old_node) {
        new_node = new tree::node(old_node->emp);
        copy(new_node->left, old_node->left);
        copy(new_node->right, old_node->right);
    }
}
tree::tree(const tree &tree) : //copy constructor
    root(0) {
    copy(root, tree.root);
}
```

- **This solution has allowed the client to create other node classes to represent other types of data without running into naming conflicts.**

Tree Implementation File

```
static void destroy(tree::node* node_ptr) {
    if(node_ptr) {
        destroy(node_ptr->left);
        destroy(node_ptr->right);
        delete node_ptr;
    }
}
tree::~~tree() {           //destructor
    destroy(root);
}
```

- **Since these utility functions are recursive in nature, each invocation relies on its arguments to determine both the results of the operation and the depth of recursion. Therefore, they are prime candidates for being considered as non-member static functions.**

Tree Implementation File

```
static tree::node* add(tree::node* node_ptr,
                      tree::node* new_node) {
    if (node_ptr) {
        if(new_node->emp.get_salary() <
            node_ptr->emp.get_salary() )
            node_ptr->left = add(node_ptr->left, new_node);
        else
            node_ptr->right = add(node_ptr->right, new_node);
        return (node_ptr);
    } else
        return (new_node);
}
void tree::insert(const employee &emp) { //insert employee
    node* new_node = new node(emp);
    root = add(root, new_node);
}
```

Static Member Functions

- By simply making the static utility functions members, they have direct access to the node class' public members, even if the node class is defined in the tree class' private section. This can be done without declaring our functions as friends.
- Static member functions do not have a this pointer, just like non-member functions.

```
class tree {  
    ...  
    private:  
        node* root;           //root node of tree  
        static void copy(node* &, const node*);  
        static void destroy(node*);  
        static tree::node* add(node*, node*);  
        static void traverse(const node*);  
};
```

Static Member Functions

```
tree::tree(const tree &tree) : //copy constructor
  root(0) {
  copy(root, tree.root);
}
```

```
//This is the implementation of a static member function
void tree::copy(node* &new_node, const node* old_node) {
  if(old_node) {
    new_node = new node(old_node->emp);
    copy(new_node->left, old_node->left);
    copy(new_node->right, old_node->right);
  }
}
```

Static Members

- Shared properties are represented by static class members. Static class members can be either static data members or static member functions. These members can be declared in either the public, protected, or private sections of a class interface.
- When a member function is declared as static, we are not limited to using this function through an object of that class. This function can be used anywhere it is legal to use the class itself. This is because static member functions are invoked independent of the objects and do not contain a this pointer.
- It can be called via: **class::function(args)**

Static Members

- To specify static member functions, we must declare the functions to be static in the class definition (interface file) using the keyword `static` .
- Then, define those static member functions in the class implementation file.
- The definition of our static member functions should look identical to the definition of non-static member functions.
- The scope of a static member function is the same as that of a non-static member function.
- Access to a static member function from outside the class is the same as access to a non-static member function and depends on whether the member is declared as `public`, `protected`, or `private`.

Static Data Members

- When we use the keyword `static` in the declaration a data member, only one occurrence of that data member exists for all instances of the class. Such data members belong to the class and not to an individual object and are called static data members. Static data represents class data rather than object data.
- To specify a static data member, we must supply the declaration inside the class interface and the definition outside the class interface. Unlike a non-static data member, a static data member's declaration does not cause memory to be allocated when an object is defined.

Static Data Members

```
//static data member declaration (class interface)  
class class_name {  
    ...  
    static data_type static_data_member;  
    ...  
};
```

- **A static data member's definition must be supplied only once and is usually placed in the class implementation file. A static data member's definition must be preceded by the class name and the scope resolution operator before the static data member's identifier.**

```
//static data member definition (class implementation)  
data_type class_name::static_data_member = initial_value;
```

Static Data Members

- It is important to realize that memory is allocated for static data members when we explicitly define those static data members, not when we declare the static data members as part of a class definition. Think of the static data member declarations within a class as external references to data members defined elsewhere.
- Clients can access a public static data member by saying `class_name::static_data_member`
- It is also possible for clients to access a public static data member once an object is defined for that class by saying `object_name.static_data_member`.

Introduction to C++



**Template
Classes**

Class Templates

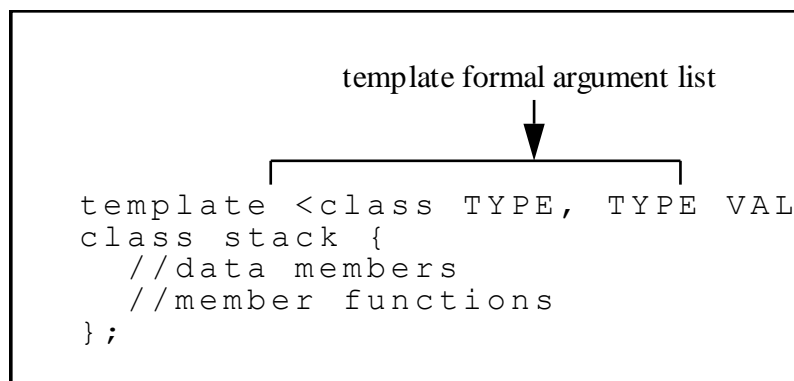
- **Class templates allow us to apply the concept of parametric polymorphism on a class-by-class basis.**
- **Their purpose is to isolate type dependencies for a particular class of objects.**
- **Using class templates, we shift the burden of creating duplicate classes to handle various combinations of data types to the compiler.**
- **Type dependencies can be found by looking for those types that cause a particular class to differ from another class responsible for the same kind of behavior.**

Class Templates

- With class templates, we simply write one class and shift the burden of creating multiple instances of that class to the compiler.
- The compiler automatically generates as many instances of that class as is required to meet the client's demands without unnecessary duplication.
- Specialized (and partially specialized) template classes can be used to handle special cases that arise.
- The syntax for implementing class templates is similar to that of defining and declaring classes themselves. We simply preface the class definition or declaration with the keyword `template` followed by a parameterized list of type dependencies.

Class Templates

- We begin with the keyword `template` followed by the class template's formal argument list within angle brackets (`< >`). This is followed by the class interface.
- The template formal argument list consists of a comma separated list containing the identifiers for each formal argument in the list. There must be at least one formal argument specified.
- The scope of the formal arguments is that of the class template itself.



Class Templates

- The template formal argument list can contain three different kinds of arguments:
 - identifiers that represent type dependencies,
 - identifiers that represent values, and
 - identifiers that represent type dependencies based on a template class.
- A type dependency allows the identifier listed to be substituted for the data type specified by the client at instantiation time. More than one identifier can be listed, each prefaced by the keyword **class** or **typename** to indicate that this is a type dependency.

Class Templates

- We can specify a non-type identifier in the template formal argument list, prefaced by its data type.
- A non-type specifier allows the identifier listed to be substituted for a value specified by the client at instantiation time.
- **template <data_type nontype_identifier>**
- The data type of non-type identifiers must be an integral type, an enumeration type, a pointer to an object, a reference to an object, a pointer to a function, a reference to a function, or a pointer to a member.
- They cannot be void or a floating point type.

Class Templates

- Clients must specify the values for non-type identifiers explicitly inside of angle brackets when defining objects class templates.

```
//function template declaration
```

```
template <char non_type, class TYPE_ID>
```

```
class t_class {
```

```
    public:
```

```
        TYPE_ID function(int TYPE_ID);
```

```
};
```

```
//client code
```

```
t_class <'\n',int> obj;
```

- Using non-type formal arguments allows the client to specify at compile time some constant expression when defining an object, which can be used by the class to initialize constants, determine the size of statically allocated arrays, or any other initialization.

Class Templates

- A class template's formal argument list may include other template classes. This means that the type dependency is based on an instantiation of another abstraction. When this is the case, the identifier representing the type dependency is preceded by the keyword `template`, the specific type dependencies to be substituted supplied in angle brackets, and the keyword `class`:
 - `template <template <actual_args> class_id>`
- When making use of this features, the template classes being used as formal arguments must be defined before the first use that causes an instantiation of the class template.

Class Templates

- Class templates declared to have default template formal arguments can be used by the client with fewer actual arguments.
- Default arguments may be specified for type dependencies, non-type values, and template class arguments.

```
template <class TYPE=int> class list;
```

- To create an instantiation of a class using all default values for initialization, empty angle brackets must be used by the client:

```
list <> object;
```

Class Templates

- Member functions that are part of a class template are themselves template functions. Therefore, we must preface member functions defined outside of the class with the template's header.
- Member functions can either be defined inside of a class template as inline members or separated from the class' definition.

```
template <class TYPE1, int sz,template<TYPE1> class
                TYPE2>
list & list<TYPE1,sz,TYPE2>::operator = (const list &) {
    //function's definition
}
```

Class Templates

- **Template classes can support static data members. A static data member declared within a class template is considered to be itself a static data member template.**
- **With template classes, the compiler automatically allocates memory for the static members for each instantiation of the class.**
- **Each object of a particular template class instantiation shares the same static data members' memory. But, objects of different template class instantiations do not share the same static data members' memory.**

Class Templates

- Such static data members are declared within the class definition and are defined outside the class, possibly in the class implementation file.

```
template <class TYPE>
  class stack {
    static int data;

  }
  template <class TYPE>
    int stack<TYPE>::data = 100;
```

Class Templates

- When defining an object, clients must specify the data types and values used for substitution with the type and non-type dependencies.
- For example, if a class template has one type dependency and one non-type:
 - **class_name <data_types, values> object;**
- Only instances used by the client cause code to be generated for any particular compilation.
- If the identifiers representing the types expected match exactly and if the non-type template arguments have the identical values as a previous instantiation, the template class is not re-instantiated.

Class Templates

- A class template will be implicitly instantiated when it is referenced in a context that requires a completely defined type.
- `class_name <int> object` causes an implicit instantiation.
- Because objects are not formed when we say `class_name <int> * ptr`; therefore, the class does not need to be defined and an integer instantiation of this class is not generated.
- However, when the pointer is used in a way that requires the pointer type to be defined (such as saying `*ptr`), then an instantiation will be generated at that time.

Class Templates

- If we define data members that are themselves template classes, the classes are not implicitly instantiated until those members are first used.
- In the following class, the `list_object` data member's class is not implicitly instantiated until the point at which data member is first used.
- On the down side, unless explicit instantiation (explained in the next section) is requested, errors can only be found by explicitly using all member functions for each use expected. This applies to syntax errors as well as run time errors.

Class Templates

```
template <class TYPE, int sz>
class stack {
public:
    stack();
    int push(const TYPE & data);
    TYPE & pop();
private:
    //this does not instantiated the list class
    list <int, 100, stack> list_object;
};
```

```
template <class TYPE, int sz>
int stack<TYPE,sz>::push(const TYPE & data) {
    //if this is the first usage of the list_object member,
    //then the list class is instantiated
    list_object <<data;
    ...
```


Class Templates

- Clients can cause explicit instantiations to take place by placing the keyword `template` in front of a class name when defining objects.
- Explicit instantiations can be used to improve the performance of our compiles and links.
- It can be used to support better code generation and faster and more predictable compile and link times.
- The drawback is that a template class can only be explicitly instantiated once, for a particular set of template actual arguments.

Class Templates

- By explicitly instantiating a class we also cause all of its member functions and static data members to also be explicitly instantiated unless they have previously been explicitly instantiated themselves.
- We can also explicitly instantiate just select member functions of a class template if all member functions are not needed.
- Explicit instantiations of a class are placed in the same namespace where the class template is defined, which is the same if the class were implicitly instantiated.
- But, default template arguments cannot be used in an explicit instantiation.

Class Templates

- And, the class template must be declared prior to explicitly instantiating such a class.

```
//class template declaration
```

```
template <class TYPE, int sz> class stack;
```

```
template class stack<int,100>; //explicit instantiation
```

- When we use a class identifier as the second operand of the direct or indirect member access operators (the . and -> operators) or after the scope resolution operator, we must precede the class' identifier with the keyword template.

```
list_object->template node<int> * ptr = new node<int>;
```

```
//the following is illegal if node is a class template
```

```
list_object->node<int> * ptr = new node<int>;
```

Specializations

- Supporting special cases is particularly important when dealing with class templates.
- For some types, we may find that certain member functions need partial specialization.
- For other types, we may find that additional data members are required.
- To support special cases, we can implement specific instantiations of our member functions and classes to customize the functionality for a given set of data types and values.

Specializations

- To specialize a member function, we must define the specialized version of the function. A partial specialization specifies what it expects as the template's actual arguments following the class identifier. These arguments must be specified in the order that corresponds to the formal template argument list.
- Not all arguments must be specialized; in these situations, the identifiers specified in the formal template argument list may be used instead of specific types and/or values. When all arguments are specialized, we have an explicit specialization; in this case, the template's formal argument list is empty (e.g., `template <>`).

Class Templates

```
template <class TYPE> class stack {
private:
    TYPE * stack_array;
    const int stack_size;
    int stack_index;
public:
    stack (int size=100): stack_size(size), stack_index(0) { stack_array =
        new TYPE[size]; } ...
};
template <class TYPE> void stack<TYPE>::push(TYPE item) {
    if (stack_index < stack_size) {
        stack_array[stack_index] = item;
        ++stack_index; } }
//An explicit specialization
template <> void stack <char *>::push(char * item) {
    if (stack_index < stack_size) {
        stack_array[stack_index] = new char[strlen(item)+1];
        strcpy(stack_array[stack_index], item);
        ++stack_index; } }
```

Specializations

- **Partial specialization of a class template is the mechanism that we can use to cause one implementation of a class to have different behavior than another.**
- **This is primarily useful when a class requires different behavior depending on the data types used by the client.**
- **We must define the specialized version of the class after the class template is defined or declared (called the primary template).**
- **All members must be completely defined for that version. This means that all member functions must be defined for a specialized class template, even in the case where the functionality is unchanged.**

Class Templates

//primary class template

```
template <class TYPE1, int sz, template<TYPE1> class TYPE2>  
  class list {...}
```

//partial specialization

```
template <class TYPE1, int sz, template<TYPE1> class TYPE2>  
  class list <TYPE1 *,sz,TYPE2> {...}
```

//partial specialization

```
template <int sz, template<TYPE1> class TYPE2>  
  class list<list, sz,TYPE2<list>>{...}
```

//partial specialization

```
template <template<TYPE1> class TYPE2>  
  class list<list,100,TYPE2<list>>{...}
```

//explicit specialization

```
template <> class list<list,100,list<list>>{...}
```


Class Templates

```
template <class TYPE> class stack {
private:
    TYPE * stack_array;
    const int stack_size;
    int stack_index;
public:
    stack (int size=100): stack_size(size), stack_index(0) { stack_array =
        new TYPE[size]; }
    void push(TYPE item);
    TYPE pop(void);
};
template <class TYPE> class stack <char *> {
private:
    char ** stack_array;
    int stack_index;
public:
    stack(int size=100): stack_index(0){
        stack_array = new char *[size]; } ...
```

Using Separate Files

- Member functions can be defined as inline in the same file as the class's interface using the inline keyword or defined in a separate file.
- When we define the member functions of our class templates in a separate implementation file, we define the class' interface in a header file in the same way that we would do for a non-template class.
- To do so requires that we define or declare our member function templates with the export keyword.
- This tells the compiler that the functions can be used in other "translation units" and may be compiled separately

Using Separate Files

- There are some restrictions.
- Templates defined in an unnamed namespace cannot be exported.
- But, an exported template only needs to be declared in the file in which it is instantiated.
- Declaring member functions of a class template as exported means that its members can be exported and used in other files.
- If the keyword `export` was not used, the definition of the function must be within the scope of where we define the objects. This would require that we include the implementation file in our header file and ultimately in client code as well (e.g., `t_class.h` could include `t_class.cpp`)

Using Separate Files

```
#include "t_class.h"
main() {
    t_class<int, float> obj; //defining an object
}
```

```
//t_class.h
//declarations of the function template(s)
template<class TYPE_ID1, class TYPE_ID2>
class t_class {
public:
    t_class();
    t_class(const t_class &);
    ~t_class();
    void t_member(TYPE_ID1, TYPE_ID2);
private:
    TYPE_ID1 data_1;
    TYPE_ID2* ptr_2;
};
```

Using Separate Files

```
//t_class.cpp
//implementation of the member function
export template<class TYPE_ID1, class TYPE_ID2>
void t_member(TYPE_ID1 arg_1, TYPE_ID2 arg_2) {...}

export template<class TYPE_ID1, class TYPE_ID2>
t_class() {...}

export template<class TYPE_ID1, class TYPE_ID2>
t_class(const t_class & source) {...}

export template<class TYPE_ID1, class TYPE_ID2>
~t_class() {...}
```

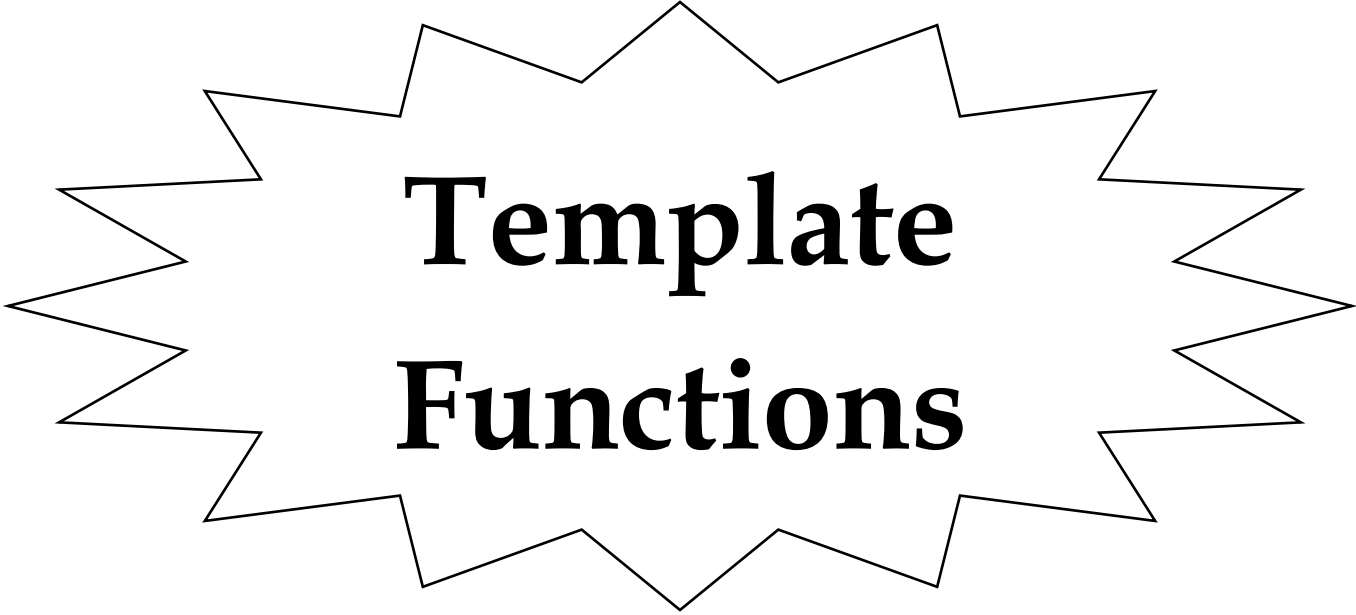
Caution

- **One of the most serious drawbacks of class templates is the danger of generating an instance of the class that is incorrect because of type conflicts.**
- **Except by writing specialized template classes to take care of such cases, there is no way to protect the client from using the class incorrectly, causing erroneous instances of the class to be generated.**
- **Be very careful when writing class templates to ensure that all possible combinations will create correct classes.**

Caution

- Realize that when we write a class template we may not know the types that will be used by the client.
- Therefore, we recommend using type dependencies only once within the class's formal argument list.
- Also, make sure to handle the use of both built-in and pointer types.
- And, when we implement our own user defined types, realize the importance of overloading operators in a consistent fashion -- so that if template classes are used with those new data types, the operators used by the member functions will behave as expected and will not cause syntax errors when used...

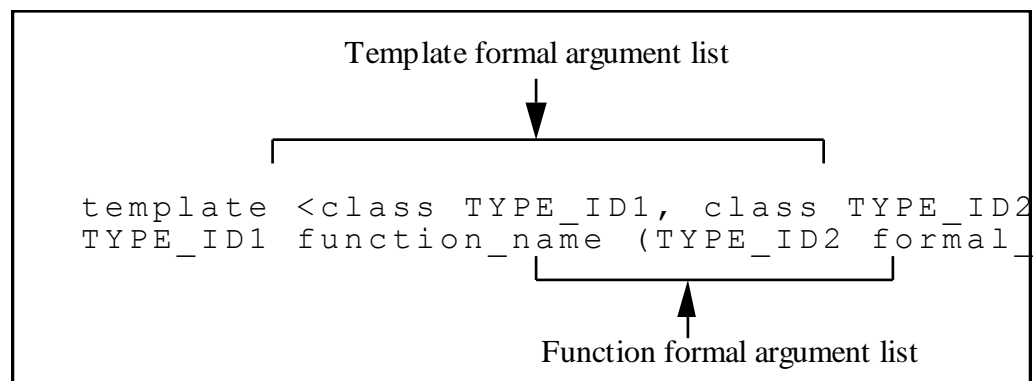
Introduction to C++



**Template
Functions**

Function Templates

- We begin with the keyword `template` followed by the template's formal argument list inside of angle brackets (`< >`).
- This is followed by the function's header (i.e., the function's return type, name, and argument list).
- The signature of a function template is the function's signature, its return type, along with the template's formal argument list.



Function Templates

//Prototype:

```
template <class TYPE1, class TYPE2>  
void array_copy(TYPE1 dest[], TYPE2 source[], int size);
```

//A later definition:

```
template <class TYPE1, class TYPE2>  
void array_copy(TYPE1 dest[], TYPE2 source[], int size) {  
    for(int i=0; i < size; ++i)  
        dest[i] = source[i];  
}
```

//The client can call this function using:

```
int int_array1[100], int_array2[100], int_array3[20];  
float real_array[100];  
char char_array[20];  
array_copy(int_array1, int_array2, 100);  
array_copy(int_array3, int_array1, 20);  
array_copy(real_array, int_array1, 100);  
array_copy(int_array1, char_array, 20);
```

Function Templates

- Type dependencies are deduced by the compiler by matching the actual arguments in a call with the formal argument in the function template formal argument list.
- Type dependencies may also be explicitly specified in the function call, adding flexibility in how we specify our formal arguments.
- By explicitly specifying type dependencies, the return type can be a type distinct from the types in the function's formal argument list.
- `function_identifier <data type list> (actual arguments)`

Function Templates

- We can explicitly specify both types, or just the first type. However, it is not possible to explicitly specify the type for the second type without specifying the type for the first type as well.

```
array_copy<int [], int []>(int_array1, int_array2, 100);  
array_copy<float []>(real_array, int_array1, 100);
```

- We can have the return type also be based on a type dependency by specifying an additional type in our template's formal argument list. Because a return type cannot be deduced, clients must specify that type explicitly when calling the function.

Function Templates

```
template <class TYPE1, class TYPE2, class TYPE3>
TYPE1 array_copy(TYPE2 dest[], TYPE3 source[], int size) {
    for(int i=0; i < size; ++i)
        dest[i] = source[i];
}
```

```
//client program
char a[100];
char b[20];
cin.getline(b, 20, '\n');
```

```
//explicitly specify type dependency for only first type
int result;
result = array_copy<int> (a,b,strlen(b));
```

Function Templates

- A specialized template function is a function that we implement with the same signature as a template function instantiation.
- The specialization must be defined after the definition of the function template, but before the first use of the specialization.
- The definition of the specialized template function must be preceded by the template keyword and followed by an empty set of angle brackets (<>) (i.e., `template<>`).
- The specialization will then be used instead of a version that the compiler could instantiate.

Function Templates

- The most specialized functions are those whose arguments can be applied to a generalized function template or some other specialization.
- When calling a function, the most specialized function is used, if one is available that matches the actual argument list.
- This means that specialization takes precedence followed by generalized function templates.
- If a regular C++ function with the same name and signature as a specialized template function is declared before the definition of the function template, that declaration is hidden by the template or any specialized template function that follows.

Function Templates

```
template <class TYPE1, class TYPE2> //function template
void array_copy(TYPE1 dest[], TYPE2 source[], int size) {
    for(int i=0; i < size; ++i)
        dest[i] = source[i];
}
template<> void array_copy(char* dest[], char* source[], int size) {
    for(int i=0; i < size; ++i) {
        dest[i] = new char[strlen(source[i]) + 1];
        strcpy(dest[i], source[i]);
    }
}
//This specialized function is called when:
char saying1[] = "Hello World";
char saying2[] = "This is a great day!";
char* s1[2] = {saying1, saying2};
char* s2[2];
array_copy(s2, s1, 2);
```


Using Separate Files

- It is possible to define our function templates in a separate implementation file and simply declare that those functions exist in the software that uses them. To do so requires that we define or declare our function templates with the `export` keyword.
- When `export` is not used, the template function must be defined in every file that implicitly or explicitly instantiates that template function.
- Functions exported and declared to be `inline` are just taken to be `inline` and are not exported.

```
//t_func.h declarations of the function template(s)
template<class TYPE_ID1, class TYPE_ID2>
void t_func(TYPE_ID1, TYPE_ID2);
//t_func.cpp implementation of the function template(s)
export template<class TYPE_ID1, class TYPE_ID2>
void t_func(TYPE_ID1 arg_1, TYPE_ID2 arg_2) {...}
```

Using Separate Files

- So when should we use export and when should we simply include our function template implementations in our client code?
- When we have large functions or when there are many type dependencies or specializations, it is often easier to debug our code by using separate compilation units and use the exporting process.
- On the other hand, if we have small function templates, few type dependencies, and limited (to no) specialized templates, then there may be no advantage to compiling them separately.

Function Templates

```
template <class TYPE> //qsort.h
inline void swap(TYPE v[], int i, int j) {
    TYPE temp;
    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
template <class TYPE>
void qsort(TYPE v[], int left, int right) {
    if (left < right) {
        swap(v, left, (left+right)/2);
        int last = left;
        for (int i=left+1; i <= right; ++i)
            if (v[i] < v[left])
                swap(v, ++last, i);
        swap(v, left, last);
        qsort(v, left, last-1);
        qsort(v, last+1, right);
    }
}
```

Function Templates

```
//main.cpp
#include <iostream>
using namespace std;
#include "qsort.h"
int ia[]={46, 28, 35, 44, 15, 22, 19 };
double da[]={46.5, 28.9, 35.1, 44.6, 15.3, 22.8, 19.4};

int main() {
    const int isize=sizeof(ia)/sizeof(int);
    const int dsize=sizeof(da)/sizeof(double);
    qsort(ia, 0, isize-1);    // integer qsort
    for(int i=0; i < isize; ++i)
        cout << ia[i] << endl;

    qsort(da, 0, dsize-1);    // double qsort
    for(i=0; i < dsize; ++i)
        cout << da[i] << endl;
    return (0);
}
```

Caution

- **One of the most serious drawbacks of function templates is the danger of generating an instance of the function that is incorrect because of type conflicts.**
- **Except by writing specialized template functions to take care of such cases, there is no way to protect the client from using the function incorrectly, causing erroneous instances of the function to be generated.**
- **Be very careful when writing function templates to ensure that all possible combinations will create correct functions.**

Caution

- Realize that when we write a function template we may not know the types that will be used by the client.
- Therefore, we recommend using type dependencies only once within the function's formal argument list.
- Also, make sure to handle the use of both built-in and pointer types.
- This may mean that we provide overloaded generalized function templates or template function specializations.