**TEMPLATES**

Recall swapValues function that only works for integers:

void swapValues(int& var1, int& var2){

int temp;

temp = var1;

var1 = var2;

var2 = temp;

}

Allow swap values of any type variables:

template<class T> --> “template prefix”, you can say typename instead of class

void swapValues(T& var1, T& var2){

T temp;

temp = var1;

var1 = var2;

var2 = temp;

}

First compiler checks the exact match for the function, then look template functions.

Template prefix tells compiler what’s coming is “template” and that T is a type parameter.

Type names are “parameters” instead of actual types.

Precise definition determined at compile-time.

Compiler produces the proper function according to arguments.

All happens during the compile time.

You may have more than one parameter for the templates:

template<class T, class R>

void swapValues(T& var1, R& var2){

This code is error because you cannot have unused template parameters.

T temp;

temp = var1;

var1 = var2;

var2 = temp;

}

T and R can be same or different.

Your produced code doesn’t know that it was a result of a templated function or normal function bc compiler produces the proper code during the compile time.

If you don’t call the templated function in your program, it wouldn’t be compiled. So if you do things like “k = y;” in the templated function without defining k and y, your code still can be compiled.

If you do “void f() { }”, doesn’t matter you call f or not, it has to be compiled and put in your .o file unlike templated function. It’s because maybe you are linking your .o file with somebody else and he/she may use your f function.

You can call swapValues function with your object from any class like DayOfYear. BUT you have to have assignment operator for that class. For example you cannot do: “int a[10], b[10]; swapValues(a, b);” bc arrays cannot be assigned.

So all the operations that you do in templated function has to be valid with the new type that you are providing.

Declaration/Prototype:

template<class T>

void showStuff(int stuff1, T stuff2, T stuff3);

Definition:

template<class T>

void showStuff(int stuff1, T stuff2, T stuff3){

cout << stuff1 << endl << stuff2 << endl << stuff3 << endl;

}

**Compiler Complications**

* Function declarations and definitions
  + Typically we have them separate
  + For templates 🡪 not supported on most compilers!
* Safest to place template function definition in file where invoked
  + Many compilers require it appear 1st
  + Often we #include all template definitions
* We have to keep template headers and template implementation in header file
* Check your compiler’s specific requirements
  + Some need to set special options
  + Some require special order of arrangemenet of template definitions vs. other file items
* Most usable template program layout
  + Template definition is same file it’s used
  + Ensure template definition precedes all uses
    - Can #include it

**CHECK main.cpp, sort.h, sort.cpp in order.**

In vector class:

template<typename \_Tp, typename \_Allocator = std::allocator<\_Tp> >

class vector : public …

…

Here we don’t use second type parameter bc it is defaulted.

**Algorithm Abstraction**

* Refers to implementing templates
* Function templates are one way C++ supports algorithm abstraction
* Express algorithms in “general” way:
  + Algorithms applies to variables of any type
  + Ignore incidental detail
  + Concentrate on substantive parts of algorithm

Actually template is like a macro:

#define SQ(x) ((x)\*(x)) 🡪 very primitive template

#define SQ(x) decltype(x) sqr(decltype(x) p){ \\

return (x)\*(x);}

#define SQ(x) x sqr(x p){\\

return p\*p;}

**Defining Template Strategies**

Develop function normally using actual data types

Completely debug “ordinary” function

Then convert to template

* Replace type names with type parameter as needed

Advantages:

* Easier to solve “concrete” case
* Deal with algorithm, not template syntax

If you have templated function f, you can force the compiler to call f with an int exactly: f<int>(2.0);

This is not nice but possible.

**Class Templates**

- String class is actually instantiation of basic\_string with a character.

Example:

template<class T>

class Pair{

public:

Pair();

Pair(T firstVal, T secondVal);

void setFirst(T newVal);

void setSecond(T newVal);

T getFirst() const;

T getSecond() const;

private:

T first; T second;

};

I can have: Pair < DayOfYear > o1, o2;

If T involves pointers

I can have: Pair < Pair<DayOfYear> > o1, o2;

Conditions for T:

* Assignment operator, copy constructor, and destructor (they have to be public)

template<class T>

Pair<T>::Pair(T firstVal, T secondVal){

first = firstVal;

second = secondVal;

}

template<class T>

void Pair<T>::setFirst(T newVal){

first = newVal;

}

Declaration of objects:

Pair<int> score;

Pair<char> seats;

Class name before :: is “Pair<T>” not just “Pair”

Constructor name is just “Pair”

Destructor name is also just “~Pair”

**CHECK 16.04.cpp.**

You can say:

template<class T = int>

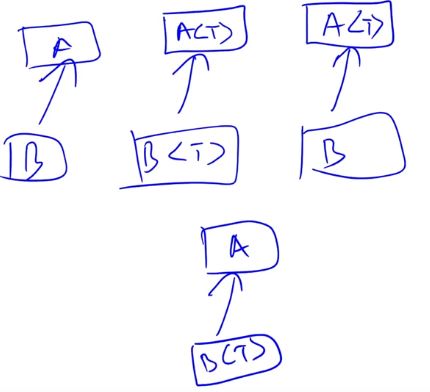
class Pair{

public:

...

So when you do this: “Pair i;” , this means you have pair of integers.

**CHECK 16-10.cpp, pfarray.h, pfarray.cpp, pfarraybak.h, pfarraybak.cpp in order.**



**You can do every inheritance you see in the left.**

class PFArrayDBak : public PFArray<double> { … };

I don’t need template type for PFArrayDBak.

class PFArrayDBak : public PFArray<T> { … };

Now I am a templated class. So you need “template<class T>” before.

int addUP(const Pair<int>& the\_Pair);

addUP is not a templated function. If I don’t include any generic type inside my (class) function header, this is not a templated function.

Template types can be used anywhere standard types can.

template<class T>

T addUP(const Pair<T>& the\_Pair);

//Precondition: Operator + is defined for values of type T

//Returns sum of two values in thePair

Now addUP is a templated function.

If you want addUP to work ONLY with int and double nothing else, you can do:

add.cpp

template<…>

addUP(…){ …

}

…

void f(){

addUP<int>(…);

addUP<double>(…);

}

//add.cpp becomes an .o file so we won’t give codes to the customer

add.h

{You only include prototypes for the int addUP and double addUP}

Whenever customers include add.h, they see only the int and double versions of addUP. Because you didn’t give any instantiation to anybody.

All they see is definition for addUP but it is intantiated with int and double.

You can do this kind of things: “typedef Pair<int> PairOfInt;” and then use “PairOfInt pair1, pair2;”

**Friends and Templates**

Friend functions can be used with template classes.

* Same as with ordinary classes
* Simply requires type parameter where appropriate.

Very common to have friends of template classes

* Especially for operator overloads (as we have seen)

f <int T> (…){

for( … )

for(…)

for(…)

if (T == 1)

a;

else

b;

}

This is better than having an int (int test) parameter. Because during the compile time, compile knows T’s value for example if it is 1, then compiler doesn’t compile b.