Brown University ENGN2912B Fall 2015
Scientific Computing in C++
Lecture 10 | Preprocessor , Templates

The `#define` preprocessor directive

The programmer can make use of the compiler preprocessor directives to define symbols and expressions. These definitions are inserted whenever the defined symbol or expression appears elsewhere in the program. The **`#define`** directive has already been encountered in the formation of the .h file *guard*. The general form is,

`#define` *identifier replacement*

The replacement value or expression is substituted for the identifier at every occurrence. An example of a value replacement is,

```
#define PI 3.14159
```

`PI` is not a constant in the sense of C++ types, it is simply a symbolic name for the number **`3.14159`**; and its type is not specified. The preprocessor substitutes the number for the symbol as follows.

```
double rad_to_deg(double x) {
  double r = 180.0/PI; //PI is just 3.14159
  return r*x;
}
```

There seems to be little difference between using **`#define`** and just declaring a global variable ,

```
const double PI = 3.14159;
```

However, in this case the type must be declared, and storage is allocated for **`PI`**. With **`#define`**, **`PI`** becomes a custom addition to the C++ language and requires no compiled program storage unless it is actually used in a program statement.

Even more powerful is the ability to **`#define`** parameterized expressions [1]. These parameterized expressions are called "macros." Here the word *macro* means larger, indicating that a small function created by **`#define`**, expands into a larger body of code.

It can be anticipated that **`rad_to_deg`** will be used widely throughout the code so it makes sense to define it to the preprocessor. The definition is as follows.

```
#define PI 3.14159
```

[1] When speaking, **`#define`** is pronounced "***pound define***."

```
#define R2D(x) x*180/PI
int main() {
   double ang = R2D(PI/4);// ang  == 45.0
 }
```

The statement

```
 double ang = R2D(PI/4);
```

Is changed by the preprocessor into the statement,

```
double ang = 3.14159/4*180/3.14159;// ang = 45
```

where the replacement for the defined symbols and expressions are directly substituted into the statement. In this case the result will be correct because **/** takes precedence over *****. On the other hand if the statement were,

```
double ang = R2D(PI+4);
```

The expansion by the preprocessor will be,

```
double ang = 3.14159+4*180/3.14159;// ang = 232.3249
```

which is obviously wrong.

A widely used convention is to use capital letters for preprocessor defined symbols. This style alerts the programmer that a variable expression is being substituted at that point.

Definitions with arguments can be very convenient shorthand. Consider the following definition,

```
#define ERROR(x) cout << "An error of type " << #x << " has occurred" << endl
```

With this definition, the main program

```
int main() {
  double ang = -R2D(PI/4);
  if(ang<0)
    ERROR(negative angle);
}
```

produces the following output.

```
An error of type negative angle has occurred
```

In this example the symbol **#x** indicates that **x** should be interpreted as a string constant for replacement. The definition **ERROR(x)** can be use to produce easily read error messages without coding the sentence context each time. The preprocessor does this automatically. In a sense, the **#define** mechanism is automatically generating code.

Note that the **#define** expression is constructed so that the terminating semicolon is missing. This convention requires the programmer to supply a terminating semicolon when the definition is used to provide the appearance of any normal C++ statement.

So far all the definitions have been on one line. For more extensive functions that won't fit on one line, a backslash '\' is used to delineate that the linefeed is to be ignored. The following example illustrates its use.

```
#define SAFE_SQRT(x, y) if(x<0) y=0;\
                        else y=sqrt(x)
```

The main program expresses this definition as follows.

```
  double x1 = -1, x2 = 1, y1, y2;
  SAFE_SQRT(x1, y1);//y1 == 0
  SAFE_SQRT(x2, y2);//y2 == 1
```

Note that the functional appearance of the **SAFE_SQRT** macro can be misleading. For example,

```
SAFE_SQRT(++x1, y1);
```

Expands to,

```
if(++x<0) y=0;\
 else y=sqrt(++x)
```

which will produce a wrong result. The compiled code will incorrectly increment x twice.

One more useful mechanism is the concatenation syntax **##**. The following definition illustrates its use.

```
#define ASN(x,i,v) x##i = v
```

In the main program,

```
  double v1, v2;
  ASN(v,1,3.0); // v1 == 3.0
  ASN(v,2,5.0); // v2 == 5.0
```

For example, the statement **ASN(v,1,3.0);** is replaced by **v1 = 3.0;**

It is easy to get carried away using **`#define`**. Ultimately such use makes code more difficult to read and debug, since the statements actually being executed don't appear in the source at the point where they are executed. Strange, hard to interpret, error messages can result.

On the other hand, the more work the programmer can get the complier to do, is less work he or she has to do by hand. Properly designed **`#define`** definitions can considerably reduce coding effort.

Conditional compilation

Another useful function of the preprocessor is its support for conditional compilation. The conditional directives are **`#ifdef, #ifndef, #if, #endif, #else and #elif`** (else if)

The scope of the conditional is delimited by the **`#if`**… and **`#endif`** pair. The source code statements within the scope are compiled when the if condition is satisfied, otherwise they are invisible to the compiler. A typical use for this capability is to turn off debug print statements when the program implementation is finally working. If a symbol, say **`debug`** is defined before the code body as follows.

```
#define debug
```

Then the following print statement will be executed.

```
double x = 5;
#ifdef debug
  cout << x << endl;
#endif
```

If **`debug`** is never defined or a subsequent statement,

```
#undef debug
```

is encountered then the **`cout`** statement is never seen by the compiler. There can be hundreds of debug statements sprinkled throughout the code and switched on and off with a single **`#define`** statement.

The **`#if`** statement has a similar behavior except the conditional is based on a constant expression following the **`#if`**. For example,

```
double x = 5;
#if 0
  cout << x << endl;
#endif
```

also eliminates the statement from compilation. If the constant is non-zero then the statement will be compiled.

This capability is very useful for switching on and off alternative bodies of code during development. Just changing the 0 to a 1 and doing a recompile accomplishes the change, rather than inserting comment marks to stop the code from compiling.

Templates

Another mechanism in C++ for making the compiler create new code and perform computations that can be done at compile time is the *template*. The idea is similar to the normal definition of the name,

A gauge, pattern, or mold, commonly a thin plate or board,
used as a guide to the form of the work to be executed;

In C++ the template is a generic function or class that is designed to operate on any data type that satisfies a predefined set of requirements. The template specifies code that should be implemented for a suitable but *unknown* type.

The template can then be *instantiated* with a specific *known* type at any point. Template instantiation consists of automatically generating code for the class (or function) with the variable type replaced by the specified type.

This mechanism is somewhat similar to the **#define** compiler directive discussed above. However, the compiler is totally responsible for deciding where to put the implementation of templated functions and classes and when to do so. With **#define** the code is inserted directly at the statement where a defined symbol or function occurs.

The template mechanism can be illustrated with a simple example. The **point** class introduced in Lecture 7, is reworked as a templated class below.

```
template <class T>
class point {
 public:
   point();
   point(const T& x, const T& y):x_(x), y_(y)
   point(T x, T y);
   T x() const {return x_;}
   T y() const {return y_;}
   void set_x(T const& x) {x_=x;}
   void set_y(T const& y) {y_=y;}
 private:
   T x_;
   T y_;
};
```

The class definition didn't change too much. Instead of the coordinates being of type

double, they are now of a variable type denoted by **T**. This template specification is *complete* since there are no other methods on **point** to implement.
The specification starts with the key word **template**, which denotes that a template is being defined. The **< >** syntax delimits the template arguments which is a comma-separated list of variable types or an *integer* type specification, each preceded by the keyword **class**[2].

In this simple example, there is only one variable type, **T**, used in the template. Keep in mind there is nothing special about the variable name, "**T**", it can be anything that doesn't conflict with existing type names.

The remainder of the class specification is the same as before, with **double** replaced by **T**. In order to apply the templated point class to a specific type, the variable T is assigned that type as in the example below.

```
#include "point.h"
int main(){
  point<double> pd(1.0,2.0);
  double xd = pd.x();
  point<int>pi(1,2);
  int xi = pi.x();
}
```

The syntax, **point<double>**, indicates that code for the point class with **T==double** should be generated. The variable of this type , **pd**, is assigned an instance of the class **point** with **double** type coordinates. In the second case new code is generated for class point, this time with **int** coordinates.
In order for the compiler to be able to generate the implementation of point for a new data type on the fly, it is necessary for `point.h` to be included at the top of the main program file and that the full generic template implementation of the class be contained in `point.h` file, as it is.

Non-member templated functions

Suppose that stream operators are to be implemented for the **point** class. To keep things simple they will not be declared as **friend** operators as in Lecture 9, but will use the public accessor methods on **point**. The stream operators are called *templated* functions[3]. Their implementation can be put in the `point.h` file since it is included wherever operations on **point** are required. The contents of `point.h` follows.

[2] There is an alternative equivalent to the template argument **class** specifier, **typename**, which is arguably more meaningful. However for historical reasons, the **typename** designation is not widely used.
[3] Note the designation **class T** is used even when defining templated functions.

```
#ifndef point_h_
...
// point class defined
...
template <class T>
inline ostream& operator<<(ostream& os, point<T> const& p) {
  os << p.x() << ' ' << p.y() << endl;
  return os;
}

template <class T>
inline istream& operator>>(istream& is, point<T>& p) {
  T vx, vy;
  is >> vx >> vy;
  p.set_x(vx); p.set_y(vy);
  return is;
}
#endif  // point_h_
```

The implementation of the operators is pretty much the same as for a non-templated case. The only difference is the preamble line,

```
template <class T>
```

which declares that the following operator is templated over variable type **`T`**.

Note that the templated form of **`point<T>`** type is used as the argument of the stream operator. A main program that uses the templated stream operators is as below.

```
#include "point.h"
int main() {
  point<double> pd(1.1,2.1);
  point<int> pi(1,2);
  cout << "point<double> " << pd;
  cout << "point<int> "    << pi;
}
```

The stream operator instances for a given specific type are generated by the compiler when they are needed, as by the expression, **`<< pi`**. A stream operator of type **`point<int>`** is required at this point and is generated by the compiler when the expression is encountered. It is emphasized that **`"point<double>"`** is just a plain string and is not a templated expression.

The output of the main program above is as follows.

```
point<double> 1.1 2.1
point<int> 1 2
```

Note that the stream operator produces different output for the two types. It should be emphasized again that there was no extra programming to get this result. The compiler took care of handling different types automatically.

Templated friends to templated classes

```
template <class T>
class point {
public:
 point();
 point(T x, T y) : x_(x), y_(y){}
 T x() const {return x_;}
 T y() const {return y_;}
private:
 friend ostream& operator << <T>(ostream& os, point<T> const& p);
 friend istream& operator >> <T>(istream& is, point<T>& p);
 T x_; T y_;
};
```

If it is desired to access the **`private`** members of a templated class with a templated function such as the stream operators. It is necessary to use the syntax indicated above. Note that the stream operators have to be explicitly declared as a templated with the **`<T>`** syntax. As in the previous case, the arguments of the stream operators must use the templated **`point<T>`** type.

The **`inline`** implementation of the stream operators in `point.h` is altered to use the private members of **`point`**. The implementation follows.

```
template <class T>
inline ostream& operator<<(ostream& os, point<T> const& p) {
  os << p.x_ << ' ' << p.y_ << endl;//note use of private members
  return os;
}

template <class T>
inline istream& operator>>(istream& is, point<T>& p) {
  is >> p.x_ >>    p.y_;
  return is;
}
```

Multiple template arguments

Templates can be defined with multiple arguments. Consider the following function that takes an array of numbers and scales them to fit into an unsigned integer range, e.g., **`unsigned char`** or **`unsigned short`**. The result is put into an output array.

```
template <class T1, class T2, unsigned S>
void scale(const T1* a, T2* b) {
  T1 min = static_cast<T1>(1e9); // Why 1e9?
  T1 max = -min;                 // What if T1 is unsigned?
  T1 maxv = 255;                 // Why 255?
  for(unsigned i = 0; i<S; ++i) {
    if(a[i]>max) max = a[i];
    if(a[i]<min) min = a[i];
  }
  T1 r = max-min;
  if(r == 0)
    for(unsigned i = 0; i<S; ++i) b[i] = static_cast<T2>(maxv);
  else
    for(unsigned i =0; i<S; ++i) {
      T1 v = maxv*(a[i]-min)/r;
      b[i] = static_cast<T2>(v); //what if T2 is signed?
    }
};
```

In this implementation, there are three template arguments. Argument type **`T1`** represents the type of numbers to be scaled, e.g. **`double`** or **`float`**. Argument type **`T2`** represents the type of the integer result. The third argument **`S`** is the size of the array being passed. It could be argued that the size could instead be specified as an argument to the function, but maybe we want an implementation that is specialized when there are only several elements in the array, say **`S<=5`**.

The code is straightforward, but some questions immediately came up in the implementation. In order to compute the range **`r`** of the input array **`a*`**, it is necessary to have some bounds on the maximum and minimum values that type **`T1`** can have. Unfortunately, type **`T1`** is not known at time of implementation of the template code. If the maximum absolute value of **`T1`** exceeds **`1e9`** then the function will not work.

Similarly the maximum value of the output type, **`T2`**, is unknown, it could be a **`unsigned char`** or even **`unsigned long`**. Indeed, there is nothing to stop a programmer from instantiating the template with a **`signed int`**! Thus the **`maxv`** of **`255`** would only apply to **`unsigned char`**. Moreover if **`T2`** is signed then the scaling expression,

```
maxv*(a[i]- min)/r; // should be maxv*(a[i]- min)/r + minv
```

is invalid since the minimum value of **`T2`** is not **`0`**, as assumed, instead it would be some negative number, such as **`minv`**.

These problems are designated with comments in the implementation above. Thus,

the following application of the templated function will fail

```
  double arr[3]={1.0,   2.0, 1.3e10};
  unsigned short bar[3];
  scale<float, unsigned short, 3>(arr, bar);
```
because both the max values of **T1** and **T2** assigned in the function implementation above are incorrect.

Numeric traits

One common solution to these problems is the definition of *traits* for a family of types. These traits can be defined by a templated class that is used by other templated functions or classes to find out properties of a variable type. Consider a new class called **traits** that has only **static** member functions. The following is implemented in `traits.h`

```
template <class T>
class traits;

template <>
class traits<float> {
  public:
    static bool is_signed(){return true;}
    static float maxval(){return 3.40282e+038f;}
};

template <>
class traits<unsigned short> {
  public:
    static bool is_signed(){return false;}
    static unsigned short maxval(){return 65535;}
};
… //more types can be specified as needed
```

The first two lines

```
template <class T>
class traits;
```

declare that a templated class exists with name **traits**. So far, there is no implementation of the class. The only thing known at this point is that it has one variable type.

The next line,

```
template <>
```

Indicates that the templated class is going to be *specialized* for a particular known type. In the implementation to follow, **T** has type **float**. The implementation of each **traits** class instantiation declares two static methods, **is_signed** and **maxval**. The methods are implemented differently for each type value that **T** can take on. Thus **traits<T>**

allows the programmer to successfully implement generic functions without knowing the specific type being specified. The modified **`scale`** function is as follows.

```
template <class T1, class T2, unsigned S>
void scale(const T1* a, T2* b) {
  T1 min = traits<T1>::maxval(), max = 0;//value from traits
  if(traits<T1>::is_signed())//traits used to determine if signed
    max = -min;
  T2 t2max = traits<T2>::maxval();//value from traits
  T1 maxv = static_cast<T1>(t2max);
  for(unsigned i = 0; i<S; ++i) {
    if(a[i]>max) max = a[i];
    if(a[i]<min) min = a[i];
  }
  T1 r = max-min;
  if(r == 0)
    for(unsigned i = 0; i<S; ++i)
      b[i] = static_cast<T2>(maxv);
  else
    for(unsigned i =0; i<S; ++i) {
      T1 v = maxv*(a[i]-min)/r; // problem if T2 is signed
      b[i] = static_cast<T2>(v);
    }
};
```

This function will now work correctly for all types specified in `traits.h`. What if a user passes in types that haven't been defined in `traits.h`? The compiler will detect the problem,

```
Error 1     error C2027: use of undefined type 'traits<T>'
```

But note that if **`T2`** is signed then the scaling expression is still wrong, since it assumes the minimum value of **`T2`** is zero. This issue will be taken up one of the homework problems.

"code bloat"

The compiler may produce duplicate instantiations of the compiled template code every time the template is used specific types in the program, even though the same types were declared in another part of the program. For example, two different classes may have a member of type **`point<double>`**. The compiler may not be clever enough to avoid instantiating multiple copies of the implementation code of **`point<double>`**, when the declaration is encountered in different file scopes.

The repeated instantiation of templated functions and classes can lead to unacceptable program size. In small programs of a few thousand lines, this "bloat" is not a real issue. But if a program of one million lines is bloated to two million lines by repeated instantiations, another alternative to automatic instantiation by the compiler is warranted.

Fortunately, just such a mechanism exists, *explicit* template instantiation. In a context where the full template implementation is specified, the following syntax, using the key word **`template`**, forces the compiler to explicitly instantiate the template at that point. For example,

```
template void scale<float, unsigned short, 3>;
```

will cause the compiler to generate code for **`T1 == float`**, **`T2==unsigned short`** and **`S==3`**.
However, it is not yet clear how explicit instantiation helps with code bloat. What if the same **`scale<float, unsigned short, 3>`**, function is needed somewhere else in another part of the code?

Fortunately C++ compilers provide an option (compiler flag) to use explicit template instantiation where the compiler promises not to automatically instantiate a template. Instead the template will be instantiated only with the **`template`** keyword expression[4]. Now that the decision of instantiating a template is in the programmer's hands, the question arises of where to do the instantiation. There are two general rules that serve as guidelines:

1. Insure that the instantiation is only done once to avoid code bloat.
2. Declare the instantiation as soon as possible, usually where the library containing the templated code is implemented. That way the code generated for the specific template arguments can be linked to any other code using that library.

This issue will be taken up again after the design of libraries has been discussed later.

Separated compilation of templates

Another issue that the previous **`scale`** example has exposed is that the implementation of templates in a .h file can soon become unwieldy. Keep in mind that the code has to be parsed by the compiler every time the .h file is included. It would be much neater and more efficient if the implementation of a templated function or templated class could be provided in separate .cxx file, just as with non-templated functions or classes. On the other hand, a template can not be instantiated unless the full implementation is visible at the point where the template arguments are specified.

Fortunately the ability to manually instantiate templates, discussed in the previous section, can be used to solve this problem as well.

Consider again the example of the templated **`point<T>`** class discussed earlier. Suppose that it is desired to implement **–** operator on **`point<T>`** to produce a vector from two points, as well as maintain the original stream operators and accessors. However, this time the implementation will be in the .cxx file.

The implementation of **vector<T>** is done in the `vector.h` file as follows.

```
template <class T>
class vector {
 public:
  vector() : dx_(0), dy_(0) {}
  vector(T dx, T dy): dx_(dx), dy_(dy){}
  T dx(){return dx_;}
  T dy(){return dy_;}
 private:
  T dx_;
  T dy_;
};
```

The new specification for **point<T>** in `point.h` follows.

```
#include <iostream>
#include "vector.h"
template <class T>
class point {
 public:
  point(): x_(0), y_(0){}
  point(T x, T y) : x_(x), y_(y){}
  T x() const {return x_;}
  T y() const {return y_;};
  vector<T> operator - (const point<T>& p);
 private:
  friend ostream& operator << <T>(ostream& os, point<T> const& p);
  friend istream& operator >> <T>(istream& is, point<T>& p);
  T x_;
  T y_;
 };
```

Everything is pretty much the same as before, except for the new **–** operator.
What changes is that a .cxx file is introduced to hold the implementation of the **–** operator and the iostream operators. The contents of `point.cxx` is as follows.

```
#include "point.h"

template <class T> // declare that the following is template code
vector<T> point<T>::operator - (point<T> const& p)//note, :: needed
{
  vector<T> v(this->x_ - p.x(), this->y_ - p.y());
  return v;
}
```

```
// global function, so no namespace operator
template <class T>
ostream& operator<<(ostream& os, point<T> const& p) {
  os << p.x_ << ' ' << p.y_ << endl;
  return os;
}

template <class T>
istream& operator>>(istream& is, point<T>& p) {
  is >> p.x_ >>  p.y_;
  return is;
}

//explicit template instantiations of point and global functions
//for data type float

template class point<float>;
template ostream& operator<<(ostream& , point<float> const&);
template istream& operator>>(istream& , point<float>&);
```

To instantiate the **float** specialization of the template, it is necessary to include this .cxx file somewhere in the program build. Then other code in the build will be able to use these instantiations.

```
int main() {
  point<float> p1(1.0, 2.0), p2(2.0, 3.0);
  vector<float> v = p2-p1;// v.dx() = 1.0, v.dy()==1.0
}
```

This approach works, and the implementation code is separated and allowed **point<float>** to be instantiated, but how does **point<T>** and the iostream operators get instantiated for other data types such as **int**? These other types could be instantiated in the `point.cxx` file just as **point<float>** and its stream operators were. However this approach creates a lot of repetitious coding and clutters up the `point.cxx` file

The problem is solved through the use of separate includes of `point.cxx` in a special .cxx files, one for each type that is anticipated to be used. To make the coding of the repeated explicit template instantiations easy a macro is created by **#define** as follows, at the end of **point.cxx**.

```
#define INSTANTIATE_POINT(T)
template class point<T>; \
template ostream& operator<<(ostream& , point<T> const&); \
template istream& operator>>(istream& , point<T>&)
```

To proceed a series of .cxx files are created as follows. The content is shown below the filename.

```
point+double-.cxx
 #include point.cxx
 INSTANTIATE_POINT(double);

point+float-.cxx
 #include point.cxx
 INSTANTIATE_POINT(float);

point+int-.cxx
 #include point.cxx
 INSTANTIATE_POINT(int);
```

These files are compiled along with the rest of your program files as shown in my CMakeLists.txt file.

```
SET(lecture_10_sources
    point.h
    point.cxx   <-classdefinition/implementation
    point+double-.cxx              <-type instantiations
    point+float-.cxxpoint+int-.cxx
    vector.h
    lecture10.cxx                  <-main program
   )

ADD_EXECUTABLE(lecture10 ${lecture_10_sources})
```

Now only one instantiation of each type of **`point<T>`** will occur, i.e., in the special .cxx files. Note the naming convention of the files is designed to indicate the templated class name and the specified type enclosed as **`+ T -`**. The purpose of this convention is to make it easy to recognize the file where a specific instantiation is made.