We have got a BST class that can hold *integers* and that allows users to specify whatever processing they want.

But, how about floating point numbers, names, or personal records?

- Solution 1: copy and modify.
- Solution 2: let the machine do the tedious job; use templates.

---

**Example 1: A Template Swap Function**

```cpp
// No machine instructions generated here. The compiler merely memorizes the template of swap functions.
template <class T>
void swap (T& x, T& y) {
    T tmp=x;  x = y;  y = tmp;
}
main () {
    int a, b;  float c, d;
    swap (a, b); // copy and paste; function
        // void swap (int, int)
        // created 'on spot' before invoked
    swap (c, d); // copy and paste; function
        // void swap (float, float)
        // created 'on spot' before invoked
} 
```
A function name in a C++ program is actually an address – the starting address of the machine instructions of the functions.

You can store such an address in a pointer, called function pointers.

Example

```cpp
double sin(double x) { ... compute sin(x) ... }
double cos(double x) { ... compute cos(x) ... }
```

```cpp
main()
{
    double a;
    double (*p) (double); // declare a pointer to functions
    // that accept a double parameter
    // and return a double value
    p = sin; // have p points to function sin()
p(a); // compute sin(a)
    // you can also use noatation (*p)(a);
p = cos; // have p points to function cos()
p(a); // compute cos(a)
}
```
Function Operator

The familiar notation \texttt{sin()} is actually an use of the function call operator, ().

- Syntax: \texttt{func\_address ( parameter list )}.
- Just as any other C++ operators, you can provide values of () operands using arbitrarily complicated expressions, as long as the expressions produce values of desired types.
- To provide the first operand, you can directly provide the address of the function (by using function name) or give a pointer.

Further Example

Let us assume that we have implemented four sinusoid functions: \texttt{sin(x)}, \texttt{cos(x)}, \texttt{tan(x)}, and \texttt{sec(x)}. Consider the following code:

\begin{verbatim}
main() {
    double (*sinusoid[4]) (double);
    // sinusoid is an array of pointers, each entry
    // pointing to a function that accepts a double
    // parameter and returns a double value
    sinusoid[0] = sin;       sinusoid[1] = cos;
    ... determine the value of i ...
    sinusoid[i](x);         // left operand of () extracted
                            // from an array
}
\end{verbatim}
**Function Parameters**

A function parameter is a pointer parameter that points to functions.

**Motive:** In the tree traversal functions, the “processing” (that is, printing) applied to each node is fixed. How can we, the designer of the BST, anticipate all the applications/processings to be used with tree traversals?

**Solution:** We cannot. Instead, let the user of the traversal functions to determine the processing of each node and pass us a pointer to the desired function.

```c
void inorder (TreeNode* p, void (*f)(int&))
{
    if (p != NULL)
    {
        inorder (p->left, f);
        f (p->data);
        inorder (p->right, f);
    }
}
```

Alternatively, you can call the function by `(*f)(p->data);`
Two Possible Node-Processing Functions

- **void print_int (int& d)** { cout << d << '\n'; }
  
  To print the whole tree, invoke
  
  \[
  \text{inorder (root, print_int)};
  \]

- **void increment (int& d)** { d++; }
  
  To increase the value of every node by 1, invoke
  
  \[
  \text{inorder (root, increment)};
  \]