Advanced Functions

Chapter 8

Overload, Default Values, Templates

Topics

- Default Arguments
  - Optional functions, and default values
- Inline Functions
  - Optimization of simple functions
- Function Overloading
  - Functions with the same names
- Template Functions
  - Sub-routines that can be applied to any data types

Default Arguments - Motivation

- Lets examine a simple problem
  - Create a function to search for a character in the array and return its position
- Simple, I start at the beginning of the array and return the position when I find the passed char
- But this function isn't always the most useful
  - Sometimes I don't want to start my search at position zero in my array

Default Arguments - Motivation

- Starting at a non-zero position
  - Simple modification I can pass my offset for the start

```
int find(char sent[], char letter, int offset)
{
   int i;
   for(i = offset; sent[i] != '\0'; i++)
   {
     if(sent[i] == letter)
        return i;
   }
   return -1;
}
```

- But if we only use the offset about 5% of our function calls it seem obnoxious to have to pass a 0 all those times

Using Default Arguments

- Use a prototype with a parameter set equal to a constant

```
int find(char sent[], char letter, int offset = 0);
```

- With the same function

```
int find(char sent[], char letter, int offset)
{
   int i;
   for(i = offset; sent[i] != '\0'; i++)
   {
     if(sent[i] == letter)
        return i;
   }
   return -1;
}
```

Using Default Arguments

```
int find(char sent[], char letter, int offset = 0);
int main ()
{
  char myArr[] = "mississippi";
  cout << "Position of i starting from position 0: ";
  cout << find(myArr, 'i') << endl;

  cout << "Position of i starting from position 5: ";
  cout << find(myArr, 'i', 5) << endl;
  return 0;
}
```

- Here we can see that we now can pass the offset or not pass it
- If we don't pass the offset it will be set to zero by default in the function

Some Rules for Default Args

- All arguments to the right of a default argument must also be default

```
int harpo(int n, int m = 4, int j = 5); // Valid
int chico(int n, int m = 6, int j); // Invalid
int groucho(int k = 1, int m = 2, int n = 3); // Valid
```

- If only a portion of the default arguments are passed, the default arguments are overridden from left to right

```
beeps = harpo(2); // same as harpo(2, 4, 5);
beeps = harpo(1, 8); // same as harpo(1, 8, 5);
beeps = harpo(8, 7, 6); // no default arguments used
```

Inline Functions Motivation

- Some functions are very simple calls
- For instance a function that calculates the volume of a cube:

```
double volumeCube(double side)
{
   return side * side * side;
}
```

- In the execution of a program, a function call causes a physical jump in the instruction memory
  - Thus function calls have an associated overhead

- For the function shown above the overhead maybe more than the operation itself

Motivation Cont.

- One Solution
  - Don't use a function
  - Repeat the code by typing it

- We have already stated why repeating of code has its own problems

- Ideally we want the convenience of functions without the associated overhead

Inline Functions

- Inline functions can provide our ideal situation but at a price

- Lets first see an example of an inline function

```
double volumeCube(double);
int main ()
{
   cout << volumeCube(2) << endl;
   return 0;
}

inline double volumeCube(double side)
{
   return side * side * side;
}
```

Inline Function

```
double volumeCube(double);
int main ()
{
   cout << volumeCube(2) << endl;
   return 0;
}

inline double volumeCube(double side)
{
   return side * side * side;
}
```

- The keyword inline in front of the function header is all that is needed to make a normal function inline
- Notice the prototype and function call does not change

What does it do?

```
double volumeCube(double);
int main ()
{
   cout << volumeCube(2) << endl;
   return 0;
}

inline double volumeCube(double side)
{
   return side * side * side;
}
```

- The keyword *inline* suggests to the compiler to copy and paste the code of *volumeCube*() to where ever it is called
- Thus during the execute there will not be any function overhead since there will be no jumping

Inline Function Strategy

- Sweet, lets just place all inline in front of all my functions!
  - This strategy is a bad idea since it will cause your final executable to be much large than it has to be

- When to use inline functions
  - If the overhead of the function call compared to the subroutine itself is high
    - Basically if the function is short
  - If performance of a given function is critical

- Notice I used the term “suggests to the compiler”. If the function is large the compiler will often ignore your suggestion.

Function Overload

- Consider the following problem
  - Write functions which will calculate the volume of a cube and a cuboid (cube without equal sides)

- Assuming all function names must be unique we will have to be clever with our function names

Functions without Overload

```
int main ()
{
  cout << volumeCube(4) << endl;
  cout << volumeCuboid(2, 4, 6) << endl;
  return 0;
}
```

- Alright maybe not that clever
- Here we have to name the functions differently though

```
double volumeCube(double side)
{
   return side * side * side;
}

double volumeCubiod(double length, double width, double height)
{
   return length * width * height;
}
```

Functions without Overload

```
int main ()
{
  cout << volumeCube(4) << endl;
  cout << volumeCuboid(2, 4, 6) << endl;
  return 0;
}
```

- Alright maybe not that clever
- Here we have to name the functions differently though

```
double volumeCube(double side)
{
   return side * side * side;
}

double volumeCubiod(double length, double width, double height)
{
   return length * width * height;
}
```

Just how smart is our compiler

- What if we named the functions the same?
- When we call the function how will the compiler tell which function we intend on using?
- If there are two functions with the same name the compiler will resolve which function to use by examining the number and type of parameters passed.
- Lets examine our previous problem

Functions with Overload

```
int main ()
{
  cout << volume(4) << endl;
  cout << volume(2, 4, 6) << endl;
  return 0;
}


double volume(double side)
{
   return side * side * side;
}

double volume(double length, double width, double height)
{
   return length * width * height;
}
```

- Two functions with the same name but different operations
- In the main it is easy for the compiler to know which function to call based on the number of parameters

Uses of Function Overload

- There are often times when you are implementing the same functionality with two functions with two different data types.

- Example 18.1
  - Write functions to capitalize an entire string for both C++ and C style strings.

Limitations

- Consider making a overloaded function for calculating area for a triangle and a rectangle:
- **Prototypes**:

```
// Calculate Rectangle Area
double area(double width, double height);
// Calculate Triangle Area
double area(double base, double height);
```

- **Function Call:**

```
// Calculate Rectangle width 2, height 1
cout << area(2.0, 1.0);
// Calculate Triangle Area base 2, height 1
cout << area(2.0, 1.0);
```

- Since the argument types are identical the above function call is ambiguous as to which function you intended on calling
- This will not compile

One More Example

- Consider the following problem, write a function to find the average of an array of numbers

- What if there are cases where I want to both find the average of an integer array and a double array?

Implementation of Function

```
int main ()
{
   int myArr[4] = {1, 2, 3, 4};
   double ans;
   ans = average(myArr, 4);
   cout << "Answer: " << ans << endl;
   return 0;
}

double average(double arr[], int size)
{
   double avg = 0;
   int i;
   for(i = 0; i < size; i++)
      avg += arr[i];
   return avg/size;
}
```

- Cleverly write a function to accept a double array and assume it will inherently handle integers
- After all integers can easily be converted to doubles without loss

Implementation of Function

```
int main ()
{
   int myArr[4] = {1, 2, 3, 4};
   double ans;
   ans = average(myArr, 4);
   cout << "Answer: " << ans << endl;
   return 0;
}

double average(double arr[], int size)
{
   double avg = 0;
   int i;
   for(i = 0; i < size; i++)
      avg += arr[i];
   return avg/size;
}
```

- This code will not compile
- Since we are working on an array/pointer typecasting will just change the interpretation of each element at a given memory position

- This would be a very bad thing
- The data types aren't even the same size in memory

Possible Solution: Function Overload

```
double average(double arr[], int size)
{
  double avg = 0;
  int i;
  for(i = 0; i < size; i++)
    avg += arr[i];
  return avg/size;
}

double average(int arr[], int size)
{
   int avg = 0, i;
   for(i = 0; i < size; i++)
     avg += arr[i];
   return avg/size;
}
```

Perils of Function Overloading

- Fantastic we now can average doubles and integers
- But what about averaging…
  - Floats
  - Characters
  - 16 Bit Integers
  - Maybe Bools?
  - …
- Looks like you got a lot of work ahead of you

Further Examination

- All the function overloads perform nearly identical operations
- It is senseless to rewrite the same code just to accommodate a different data type
- Luckily we don't have to with template functions
- This is done through generic data types

A Template Function

```
template <typename T>
double average(T arr[], int size)
{
   T sum = 0;
   int i;
   for(i = 0; i < size; i++)
      sum += arr[i];
   return sum/((double)size);
}
```

- Template <class T>: declares generic variable to be used as a placeholder in the function below
- Notice I used *T* as a parameter for my array and as my sum variable
  - If I sum a bunch of T variables the result should also be of type T

Using our Template Function

```
template <typename T>
double average(T[], in);
int main ()
{
  int myArr[4] = {1, 2, 3, 4};
  double avg;
  avg = average(myArr, 4);
  cout << "Average is: " << avg << endl;
  return 0;
}
```

- Call the function using the normal syntax
- Notice the prototype will require another definition of `template <typename T>`
- In this example a function accepting an integer array is used

How does it work?

- Consider the idea of a template in terms of arts and crafts
  - Lets consider that your gram gram has a template for knitting you a snazzy holiday sweater
  - The template is not really a sweater in and of itself but rather an outline as to how to make this sweater

- The same logic follows for template functions
  - The template function you wrote was not directly compiled into your program
  - Confused yet? Lets examine our previous example.

Template Function

```
template <typename T>
double average(T arr[], int size)
{
   T sum = 0;
   int i;
   for(i = 0; i < size; i++)
      sum += arr[i];
   return sum/((double)size);
}
```

Function Call:

```
int myArr[4] = {1, 2, 3, 4};
double avg;
avg = average(myArr, 4);
```

- Our template function is shown above along with the call that appeared in our main function
- The template shows how to **generate** an *average* function given any data type for the first argument

The Complied Template Function

```
double average(int arr[], int size)
{
   int sum = 0;
   int i;
   for(i = 0; i < size; i++)
      sum += arr[i];
   return sum/((double)size);
}
```

Function Call:

```
int myArr[4] = {1, 2, 3, 4};
double avg;
avg = average(myArr, 4);
```

- Therefore the function generated and subsequently compiled will effectively be like the one shown above.
  - You never actually see this code though

Template Function

```
template <typename T>
double average(T arr[], int size)
{
   T sum = 0;
   int i;
   for(i = 0; i < size; i++)
      sum += arr[i];
   return sum/((double)size);
}
```

```
Function Call:
int myArr[4] = {1, 2, 3, 4};
int myArr2[3] = {1, 2, 3};
double myArr3[2] = {1.1, 2.2};

double avg;
avg = average(myArr, 4);
avg = average(myArr2, 3);
avg = average(myArr3, 2);
```

- The code to the right will cause the generation of two separate functions.
  - One to handle integers
  - One to handle doubles
- For each new data type our template function will generate a new function

Some Rules

- For each new template function, `template <typename T>` must appear right above it. (*T* can be anything, this is how you refer to your generic data type)

- It is possible generate multiple generic types `template <typename T, typename Z>`
  - This allows to variables to be passed that aren't necessary of the same data type
- Often you will often see the code below:
  - `template <class T>`
  - In this case there is no difference between *class* and *typename*
  - I will use typename since it gives better context

More Rules!

- Since a template function is not really a function some special considerations must be taken in multi-file compilation
  - The template function must go into the header file itself

- Placing the template function in the *.cpp file and prototyping it in the header will cause a linker error during compilation

Quick Examination of Header Rule

- All *.cpp files are compiled seperately
- When the *.cpp file with the template function is compiled no functions will be made
- When another file includes the header there maybe a prototype for the generic function but no template itself
- Therefore much like a structure the template function must be in the header file
  - Exception being if the template function is not to be used outside the *.cpp file

Final Note

- Templates may seem cryptic and hardly applicable but I would argue the following:

- The two most important innovations in C++
  - Classes (object orientation)
    - Technically you have done some but we have only scratched the surface
    - If you take ECE 321 you will work extensively with them
  - Templates

Why are Templates so Important

- The old C library depends heavily on overloaded functions
  - Ever wonder why you need to typecast your int to a double when you use the pow() function?
  - This made the C library very inflexible
- C does not have the capability for generic data types and here are the consequences of this:
  - Simple functions such as swap cannot exist in the standard lib.
  - Advanced data structures such as linked lists cannot exist in the standard lib?
    - The standard lib would have no way of storing generic data at each node

The wonders of Templates

- Templates have allowed for standard advanced data structures
  - Vectors: managed dynamic array
  - Lists: managed link list
  - Queues: managed FIFO
  - Stacks: managed FILO
- Even standard algorithms
  - Sorting Algorithms
  - Search Algorithms

- In practice these are enormous time savers!!!

Example 18.2

- Create a template function which swaps two variables of any type
  - Test the function using integers and chars

Example 18.3

- I have included a linked list library that I have implemented demonstrating the usefulness of templates.
- The linked list can store any data type
- It is implemented as a class which is similar to a structure
- This example is for your curiosity
  - You will not be tested on the material in this example