CPSC320

Tutorial 6

Robert Pringle

November 22, 2007

Exception Handling

- Generally exceptions are used when an unexpected and possibly erroneous event occurs.
- Exceptions that occur as a direct result of actions that are taken by the program are defined as synchronous exceptions while those that occur as a result of something outside of the program are refered to as asynchronous.
- When an exceptions is thrown or raised it is nesscary to find an exception handler that can do something about the exception. Searching for the appropriate exception handler is done as follows:
  - Look in the current block where the exception was raised.
  - If there is no handler in the current block look at parents blocks until we reach the block encompassing the scope of our current function/procedure.
  - If there is no handler within the function/procedure itself look at block of the caller evaluating blocks from where procedure/function was called. Perform the same steps for the caller as for the local procedure (including looking at its caller).

Exception Handling

- When an actual exception is thrown and the handler goes through callers to determine an appropriate handler it is usually nesscary to perform call or stack unwinding to provide the appropriate environment for the procedure/function that may contain the handler.
- After an exception has been handled it may be possible to continue the execution of the program, where we continue from depends on the exception handling mechanism.
  - Resumption model of exception handling returns control just after the point where exception occured and most often required that the stack be restored to what it was at that point.
  - Termination model of exception handling returns control just after the point where the exception was handled.

Exception Handling Practice

For the following program show the contents of the stack, excluding temporaries, before and just after the exception **Error** is handled for an exception handler using the resumption and termination models. Assume that **Error** is already defined.

```
int i=0;

void callTwo() {
 if(i == 0)
   throw new Error();
}

void callOne() {
  try {
    callTwo();
  }
  catch(Error err) {
    cout << "Some sort of error occured." << endl;
  }
}

int main() {
  try {
    callOne();
  }
  catch(...) {
    cerr << "Default Handler: Exiting" << endl;
  }
}
```

Parameter Passing Methods

- There are a number of different ways to pass parameters to procedures, these include:
  - Pass by value where the actual parameter's value is copied to another memory space. The actual parameter is not affected.
  - Pass by result where the value of the formal parameter is copied back to the actual parameter when the procedure returns.
  - Pass by value-result where the actual parameter's value is copied to another memory space and copied back after the procedure is finished.
  - Pass by reference where the formal parameter refers the memory location of the actual parameter.
  - Pass by name, which used lexical substitution of the formal parameter with the actual parameter and placement within the caller's body.

Pass by name

- Pass by name is slightly different from the other parameter passing methods in that an actual lexical substitution is used for this method.
- Pass by name is performed by the following steps:
  - Replace instances of the formal name parameter with the actual parameter, surrounded in parenthesis in the case of an expression, in the procedure body.
  - Perform lexical substitutions for local variables that conflict variables bound in the caller's scope.
  - Perform lexical substitutions so that free variable's in the procedure body are still valid even if a substitution occurs in such a way that the particular variable is no visible once the substitutions occurs.
  - Place the modified body at the place where the procedure was called.

Parameter Passing Practice

Consider the following code and determine the resulting output for pass by value, result, value-result, reference and name.

```
int g = 20;

void add(int b, int p) {
    b = g;
    p = b+g;
    return;
}

int main(void) {
    int g = 1, h=2, i=3;
    add(h,i);
    cout << h << i << endl;
    return 0;
}
```

Parameter Passing Practice

Consider the following code and determine the resulting output for pass by value, result, value-result, reference and name for the argument **arr** with the average function. Assume that the offset function is pass by value and uninitialized integers will automatically be set to zero.

```
int avg[4] = {1,2,3,0};

void average(int arr[],int offset) {
  int index = 0,total = 0;
  avg[offset] = 0;
  while(arr[total++] > 0);
  index = (--total)-1;
  while(index >= 0) {
    avg[offset] = avg[offset] + arr[index];
    arr[index--] = 0;
  }
  avg[offset] = avg[offset] / total;

}

int main() {

 average(avg,2);

 for(int index=0; index < 3; index++)
   cout << avg[index] << " ";
  cout << endl;

}
```