Advanced C++

Exception Handling

Topic #5

Exception Handling

- **Throwing an Exception**
- **Detecting an Exception**
- **Catching an Exception**
- **Examine an Example using Classes and Operator Overloading**

Exception Handling

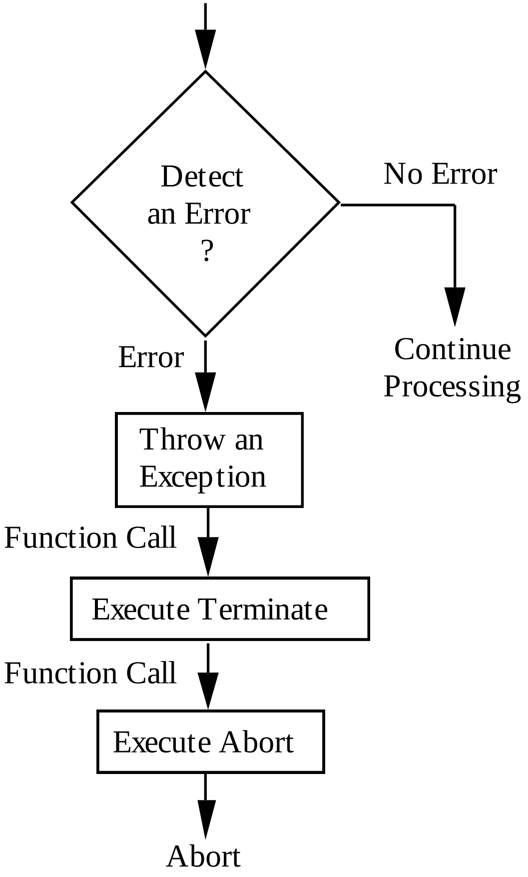
- **C++ allows us to detect error conditions at any point in a program (the throw point) and then transfer control and information (the exception) to another point in the program (the exception handler) for error processing.**
- **This process (exception handling) allows functions to detect error conditions and then defer the processing or handling of those error conditions to a direct or indirect caller of those functions.**
- **Exception handling allows us to separate normal processing from error processing.**
- **This, in turn, improves the structure, organization, and reusability of our software.**

Throwing an Exception

- **When an error condition is detected, an exception can be created and control transferred to an exception handler by executing a throw expression.**
- **A throw expression consists of the operator throw optionally followed by an operand of some type.**
  ```
  if (i != 42)  //detect error condition
  throw i;   //throw an exception
  ```
- **This causes the program to abort whenever an exception occurs. The abort occurs because the default in C++ is to abort whenever an exception is thrown that is not explicitly processed by the program. This is probably only useful for the simplest programs.**

Throwing an Exception

- An exception that is thrown and is not directly detected and handled by the program results in a call to a run time library function called terminate. The default behavior for terminate is to call abort.



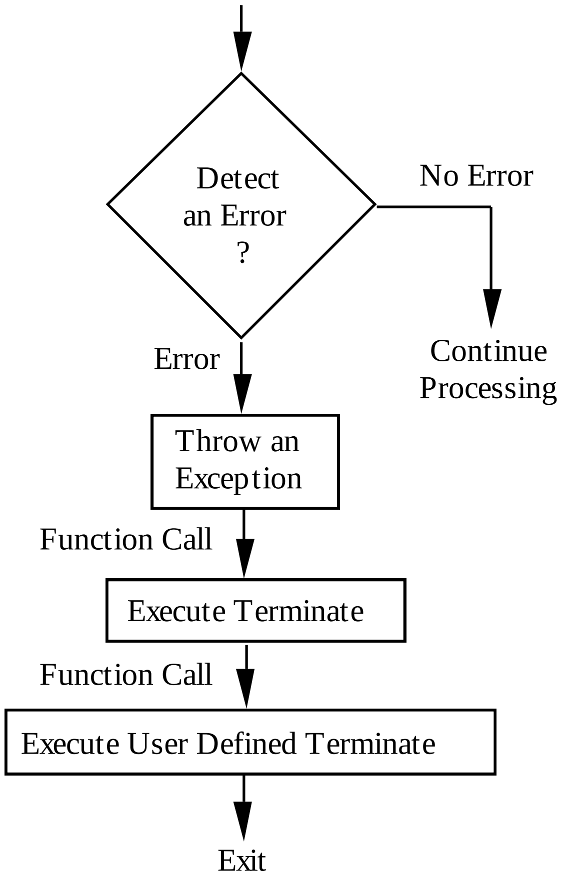
Throwing an Exception

- **Fortunately, our program can gain control by replacing the call to abort with a call to a function that we provide.**
- **We pass the address of our function to the library function set_terminate. The address of the previous function is returned and the address we pass is saved in its place.**
- **Whenever terminate is called, our function will also be called.**

```
void user_terminate() {...)      //user supplied terminate

int main() {
  set_terminate(user_terminate); //install user function
    ...
```

Throwing an Exception

- **There are four rules that apply to the function that we supply:**
  1) it must not take any arguments,
  2) it must not return data,
  3) it must not return; it can only terminate by calling exit or abort, and
  4) it is not allowed to throw an exception.
  5) the header file <exception> is needed to use set_terminate.



Throwing an Exception

```
void user_terminate() {          //user supplied terminate
  cout <<"user terminate function calling exit" <<endl;
  exit(1);                       //abnormal program exit
}

int main() {
  int i;
  set_terminate(user_terminate); //install user function
  cout <<"Enter an integer: ";
  cin >>i;

  if (i != 42)                   //detect error condition
    throw i;                     //throw an exception
  cout <<"no throw was executed" <<endl;

  cout <<"normal program exit; i = " <<i <<endl;
  return(0);
}
```

Detecting an Exception

- **To detect specific exceptions, we must specify when we want exception detection to be active.**
- **Think of this as "turning on" exception handling for a particular section of code (the try block).**
- **A try block is a compound statement preceded by the try keyword.**
- **Once the thread of control enters a try block, exception handling is in effect until the try block is exited.**
- **Think of a try block as specifying when we want to detect exceptions. We can only detect an exception that is thrown when the thread of control is inside of a try block.**

Detecting an Exception

- **Therefore, try blocks establish when exception handling is in effect.**
- **If an exception is thrown outside of a try block, terminate is called.**
- **When we are in a try block, we are able to select and handle different types of exceptions. This is called catching an exception.**
- **The following demonstrates a try block that "turns on" exception handling for our entire program:**

```
int main() {
 try {
   ...        //program code
 }
 ...          //code to handle exceptions
 cout <<"normal program exit" <<endl;
 return(0); }
```

Catching an Exception

- **When an exception is thrown, we can associate an operand of some type with the throw operator.**
- **The type of the operand determines the type of the exception.**
- **Control is passed to a block of code (the catch block) corresponding to that type.**
- **The value of the operand (the exception) associated with throw is passed to the catch block (the exception handler) as a temporary.**

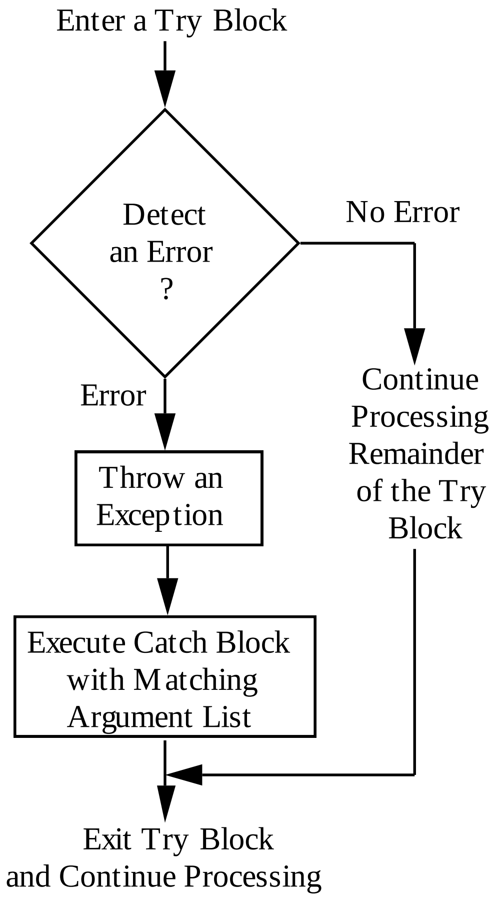
```
throw i;  //under some condition throw an exception
          //with an integer argument

catch(int x) { //catch integer exceptions
  ...
}
```

Catching an Exception

- **A catch block has access to the value of the exception, but cannot modify the original value.**
- **This is true even if the operand is a reference. Any change to the operand does not affect the original value, only the temporary copy.**
- **The type of a throw expression is void and has no residual value.**
- **If we use throw within a try block without an operand, a catch block is not executed. Instead, terminate is called.**

Catching an Exception

- **A catch block immediately follows the try block and begins with the catch keyword followed by the type and formal argument (in parentheses) that this catch block is designed to accept.**
- **There must be at least one catch block immediately following every try block.**



Catching an Exception

```
int main() {
  int i;
  cout <<"Enter an integer: ";
  cin >>i;

  try {
    if (i != 42) //detect error condition
      throw i;   //throw an exception
    cout <<"no throw was executed" <<endl;
  }

  catch(int x) { //catch integer exceptions
    cout <<"exception handler called; arg = " <<x <<endl;
    i = 42;      //set it to correct value
  }

  cout <<"normal program exit; i = " <<i <<endl;
  return(0);
}
```

Catching an Exception

- **First, a catch block is only executed as a result of throwing an exception within a try block.**
- **Second, if a throw is executed, control is immediately passed to the appropriate catch block.**
- **The statements following the throw are not executed.**
- **Third, when the catch block is done executing, control goes to the statement immediately following the try block and associated sequence of catch blocks in which the exception was handled; it does not continue with the statement following the throw.**
- **Of course, if a catch block contains a return, exit, or abort, then the program either returns from the function containing the catch block or exits the program.**

Catching Different Types of Exceptions

```
int main() {
  int i;
  cout <<"Enter an integer: ";
  cin >>i;

  try {
    if (i != 42) //detect error condition
      throw i;   //throw an exception
    cout <<"no throw was executed" <<endl;
  }

  catch(int x) { //catch integer exceptions
    cout <<"exception handler called; arg = " <<x <<endl;
    i = 42;      //set it to correct value
  }

  cout <<"normal program exit; i = " <<i <<endl;
  return(0);
}
```

Catching an Exception

- **Once a throw is executed, control is immediately transferred from the try block to the first catch block in the thread of control whose type matches the type of operand associated with the throw.**
- **The catch block is then executed and we either terminate, return, or continue at the first statement following the sequence of catch blocks in which the exception was handled.**
- **Think of throw as analogous to a function call and catch as analogous to a function definition. There can be multiple catch blocks each with a unique "formal argument" type.**

Catching an Exception

- **When we throw an exception, it is as if we are using the throw operator and its associated operand to "call" a catch block "passing" an argument. The type of the operand must be an exact match with the type specified in the associated catch block, except for the following three cases:**
  1) the operand (or, actual argument) matches a constant of the same type, a reference of that type, or a constant reference of that type,
  2) the operand is an array containing elements of some type that matches a pointer to that type, or
  3) the operand is a pointer that matches a pointer to void.

Catching an Exception

- **The operand of the throw operator determines which catch block is selected by matching its type with the type of the catch blocks.**
- **If we use the type void* for a catch block, it should come after any other catch block specifying a pointer type.**
- **This is because the catch blocks are checked in sequence and the first catch block matching the type is used.**
- **Since void* matches all possible pointer types, we would never be able to access a catch block with a specific pointer type if a void* catch block preceded it.**

Catching an Exception

- **In the previous example, we did not use the value of the throw operand in the catch blocks so we did not need to specify an identifier in the catch block's argument list, just the type.**
- **If we want to catch any exception independent of the type, we can use the ellipses (...) as the type of a catch block.**
- **If a catch block using ellipses is specified, it must be the last catch block in the sequence and is guaranteed to catch exceptions of any type.**
- **Of course, if a catch block for a particular type precedes a catch block using the ellipses, then it will catch an exception of that particular type before the catch block with the ellipses.**

Catching an Exception

```
try {
  if (c != 'x')
    throw c;  //throw exception of type char
  if (i != 42)
    throw i;  //throw exception of type int
  cout <<"no throw was executed" <<endl;
}

catch(char) { //catch char exception
  cout <<"char exception handler called" <<endl;
}

catch(...) {  //catch all other exceptions
  cout <<"universal exception handler called" <<endl;
}
```

Nested Try Blocks

- **Try blocks can be nested, either statically at compile time or dynamically based on the flow of control.**
- **When we throw an exception, the catch blocks associated with the try block containing the throw are searched for a type matching the exception.**
- **If no match is found, the catch blocks associated with the statically or dynamically surrounding try block (i.e., a try block entered, but not exited) are searched.**
- **This process continues until either a matching catch block is found or there are no more try blocks, in which case the function terminate is called.**

Nested Try Blocks

```
try {          //outer try block
  cout <<"Enter a character and an integer: ";
  cin >>c >>i;

  try {        //inner try block
    if (c != 'x')
      throw c;  //throw exception of type char
    if (i != 42)
      throw i;  //throw exception of type int
    cout <<"no throw was executed" <<endl;
  }
  catch(char) { //inner catch block
    cout <<"char exception handler called" <<endl;
  }
  cout <<"either char exception or no throw" <<endl;
}
catch(...) {    //outer catch block
  cout <<"universal exception handler called" <<endl;}
```

Nested Try Blocks

- **When an exception occurs and either: (1) no try block statically surrounds the throw point or (2) no catch blocks are found that match the type of exception thrown when a try block is present, then a process called stack unwinding begins.**
- **If we are in a function, any automatic variables and formal arguments on the stack are destroyed in the same way as when control returns from a function.**
- **But, instead of returning control to the calling function, that function is searched for a dynamically surrounding try block.**
- **If found, its associated catch block(s) are checked for a type matching the exception.**

Nested Try Blocks

- **If found, control is passed to the catch block matching the type of the exception.**
- **If no try block is found or if no catch block with a type matching the exception is found, this process of returning from a function and unwinding the stack continues until a catch block of the appropriate type is found.**
- **If none is found, the function terminate is called.**
- **The catch block itself can throw an exception in two ways:**

1) **by throwing an exception with an associated operand of some type, or**
2) **by throwing an exception with no operand (called re-throwing the exception).**

Exception Specifications

- **In order to completely declare a function, we must specify the types of exceptions that may be thrown by that function in addition to the formal argument types and return type.**
- **By default, a function can throw any exception.**
- **We can specify exactly what types of exceptions a function may throw by listing them in the function prototype or in the function header when a function is defined.**
- **This is called an exception specification.**

Exception Specifications

- **An exception specification consists of the throw keyword followed by a list of exception types enclosed in parentheses.**
- **The exception specification is the last item in a function declaration or function header.**
- **The list may be empty.**
- **This guarantees that the function will not throw any exceptions.**
- **A non empty list guarantees that the function will only throw exceptions of the type(s) specified in the list.**
- **If the exception specification is absent, the function can throw any type of exception.**

Exception Specifications

```
void a_function() throw(char, int); //exception spec

int main() {
 try {        //detect exceptions (in main)
    a_function();
    cout <<"returned from a_function" <<endl;
 }
 catch(char) { //catch char exceptions
    cout <<"char exception handler called" <<endl;
 }
 catch(int) {  //catch int exceptions
    cout <<"int exception handler called" <<endl;
 }
  cout <<"normal program exit" <<endl;
  return(0);
}
```

Exception Specifications

```
void a_function() throw(char, int) { //exception spec
  int i;
  char c;
  cout <<"Enter a character and an integer: ";
  cin >>c >>i;

  if (c != 'x')
    throw c;    //throw char exception
  if (i != 42)
    throw i;    //throw int exception

  cout <<"no throw was executed" <<endl;
}
```

- **If a function throws an exception that is not in the exception specification list, a call to a run time library function called unexpected is made. The default behavior for unexpected is to call terminate.**

Exception Specifications

- **Fortunately, our program can gain control by replacing the call to terminate with a call to a function that we provide. We pass the address of our function to the library function set_unexpected. The following shows the syntax:**

```
void user_unexpected() {       //user supplied function
  ...
}

int main() {
  set_unexpected(user_unexpected); //install user
    function
```

Catching Exceptions with new

- **When new cannot allocate the requested memory, it calls a default callback function from the standard library called new_handler.**
- **This function throws an exception of type bad_alloc.**
- **By registering a callback function, our own can be called instead of new_handler.**
- **This can be done by calling the function set_new_handler.**
- **This function takes one argument: a pointer to a function that takes no arguments and returns void.**
- **It returns a pointer to the previous callback, which is initially the default (new_handler).**

Catching Exceptions with new

```
void out_of_mem();            //user new handler callback

int main() {
  set_new_handler(out_of_mem); //install user new handler
  while(true)
    int *p = new int[1024];    //gobble up memory till gone
  return (0);
}

void out_of_mem() {
  cout <<"programmer supplied new handler called" <<endl;
  //free up space & return, throw bad_alloc, abort, or exit
  exit(1);
}
```

Exceptions w/ Classes

- **Exception handling can enhance the behavior of a user defined data type (such as adynamic array) by performing error checking.**
- **Errors can occur when new allocates memory or when an index into the array is out of bounds.**
- **Errors can be handled in two ways, either by displaying an error message and continuing processing or by throwing an exception.**

Exceptions w/ Classes (version1)

```
class dyn_a1 {
  public:
    explicit dyn_a1(INDEX) throw(); //1D array of size i
    ~dyn_a1() throw();              //destructor
    int &operator[](INDEX) throw(); //subscript operator
  private:
    dyn_a1(const dyn_a1 &);           //prohibit copy ctor
    dyn_a1 &operator=(const dyn_a1 &); //prohibit assign
    INDEX d1;              //# of elements in 1D array
    int* a0;               //base address of all elements
    int dummy;             //for out of bounds reference
};
```

Exceptions w/ Classes (version1)

```
//Implementation of dyn_a1 constructor and destructor
inline dyn_a1::dyn_a1(INDEX i) throw() :
  d1(i),                    //# of 1D array elements
  dummy(0) {
  a0 = new(nothrow) int[i]; //total # elements for 1D array
  if (a0 == 0) {            //check if new failed
    cerr <<"new failed in class dyn_a1" <<endl;
    d1 = 0;                 //set # elements to zero
  }
}
inline dyn_a1::~dyn_a1() throw() {
  delete[] a0;              //deallocate all array elements
}
//Implementation of subscript operator
inline int &dyn_a1::operator[](INDEX i) throw() {
  if (i<0 || i>=d1) {       //check if out of bounds
    cerr <<"out of bounds at index " <<i <<endl;
    return (dummy);         //reference to dummy element
  }
  return (a0[i]);           //ith element in 1D array
}
```

Exceptions w/ Classes (version2)

```
struct bad_index {          //bad index exception type
  long index;
};

class dyn_a1 {
  public:
    explicit dyn_a1(INDEX) throw(bad_alloc); //constructor
    ~dyn_a1() throw();                       //destructor
    int &operator[](INDEX) throw(bad_index); //subscript op
  private:
    dyn_a1(const dyn_a1 &);                  //prohibit copy ctor
    dyn_a1 &operator=(const dyn_a1 &); //prohibit assign
    INDEX d1;              //# of elements in 1D array
    int* a0;               //base address of all elements
};
```

Exceptions w/ Classes (version2)

```
//Implementation of dyn_a1 constructor and destructor
inline dyn_a1::dyn_a1(INDEX i) throw(bad_alloc) :
  d1(0) {                //set to 0 in case of exception
  a0 = new int[i];         //total # elements for 1D array
  d1 = i;                //# of 1D array elements
}
inline dyn_a1::~dyn_a1() throw() {
  delete[] a0;             //deallocate all array elements
}

//Implementation of subscript operator
inline int &dyn_a1::operator[](INDEX i) throw() {
  if (i<0 || i>=d1) {      //check if out of bounds
    bad_index e;           //create bad_index object
    e.index = i;           //save bad index
    throw(e);              //throw bad_index exception
  }
  return (a0[i]);          //ith element in 1D array
}
```

Exceptions w/ Classes

- **This change requires that the client program catch and handle the error and determine how to handle it.**
- **The class no longer performs error processing.**
- **This approach gives the client program control of how errors are handled and the type of error messages provided.**
- **To detect when new fails, we use the regular form of new. If new cannot allocate the necessary space, it automatically throws an exception of type bad_alloc. If the index is out of bounds, we throw an exception instead of returning.**
- **Therefore, we do not need a dummy integer to return. But, we do need to define a type for the exception that is thrown (bad_index structure).**

In Summary

- **Exception handling is difficult to do well.**
- **The exception handling facilities of C++ provide mechanisms to separate error detection from error processing.**
- **This can significantly improve the organization and reusability of our software.**
- **However, it is not a substitute for careful design.**
- **Designing software must include considering both normal processing and error processing.**
- **When exception handling is poorly used, it can create more problems than it solves by creating a false sense of security.**
- **On the other hand, if used properly, it can improve the robustness, maintainability, and reusability.**