- **Lecture Outline - Basic Exceptions**
- **Intro**
    - **how to do better error handling**
    - **standard programming problem**
        - *if ignore possibility of errors, programs crash, fail, become hard to use*
        - *but trying to detect and handle errors greatly complicates the code*
        - *every single time something might go wrong, have to check for it*
        - *AND every function that calls a function in which something could go wrong has to deal with it again*
    - **common traditional structure:**
        - *each function returns a code to say whether there is a problem*
        - *each function call must check the flag to be sure everything is OK*
        - *either way, return an error code to the caller*
            - Example sketch code using OK/Not-OK return codes

```
int main ()
{
        ....
        if f1(....) {
                cout << "error" << endl;
                do something
                }
        return 0;      // the return code!
}

bool f1 (.....)
{
        .....
        if (f2(....))
                return true;
        ....
        return false;
}

bool f2 (.....)
{
        .....
        if (f3(....))
                return true;
        ....
        return false
}

bool f3 (.....)
{
        .....
        if (z < 0)
                return true; // something's wrong!
        ....
        return false;
}
```

- **disadvantage:**
  - *lose use of return value (or have to do something even more clunky)*
  - *if' - return all over the place*
- **yuch - there's got to be a better way**
- **sometimes, there is no value to return:**
  - *Smart_array subscripting operator:*
    - See example:
      ```
      // overload the subscripting operator for this class
      int& operator[] (int index)
      {
              if ( index < 0 || index > size - 1) {
                      // this is simple, but there are better actions possible
                      cerr << "Attempt to access Smart_Array with illegal index = "
                              << index << endl;
                      cerr << "Terminating program" << endl;
                      exit(EXIT_FAILURE);
                      }
              return ptr[index];
      }
      ```
    - terminate the program because nowhere to return or check the value:
      - Smart_array a(20);
      - a[21] = 5;
      - what do we check?
      - if(a[21]) ?
- **what to do?**
- **GENERAL IDEA: PROVIDE A SEPARATE FLOW OF CONTROL FOR ERROR SITUATIONS**
  - *most of code can be written as if nothing would go wrong*
  - *separate flow of control if something does*
  - *Exception concept - an fairly old idea, developed and refined before in e.g. LISP, later C++*

- **Exception Concept**
  - **Basic syntax:**

    -
      ```
      class X {
              ... whatever you want
              };
      try {
              bunch of statements
              somewhere in here, or in the functions that are called:
                      throw X(); // create and throw an X object
              }
      catch (X& x) { // catch using a reference parameter is recommended
              do something with an X exception
              }
      ... continue processing
              e.g. try again
      ```
  - **What happens**
    - *Function calls proceed normally*
    - *but if a "throw" is executed*
      - stack is "unwound" back up to try block that is followed by the matching catch
      - the catch block is executed
      - execution then continues after the final catch
    - *unwinding the stack is equivalent to forcing a return from the function at the point of the throw, and for every function in the calling stack up to the try block*
    - *control is transferred from the point of the throw to the matching catch, with all functions in between immediately returning*
- **Sketch example**
  - **Separate error flow of control now cleans things up!**
    - *No need to tediously check return values!*
    - *Return values can now be used for the real work!*
    - *Compare to return-code sketch*

      -
        ```
        class X {
                ... whatever you want the exception class to have in it
                };

        int main ()
        {
                ....
                try {
                        ...
                         a= f1(...);
                        ...
                        }
                // catch block is ignored if no throw
                catch (X& x) {
                        cout << "error" << endl;
                        do something
                                could quit
        ```

```
                    could change values
                }
            ... continue if desired
        }

        int f1 (.....)
            {
                .....
                b = f2()
                return i; // get return values back!
            }

        int f2 (.....)
            {
                .....
                b = f3()
                return i; // get return values back!
            }

        int f3 (.....)
            {
                .....
                if (z < 0)
                    throw X(); // something's wrong!
                return i; // get return values back!
            }
```

- **Can have more than one kind of exception**
    - **Declare them, then catch them**

        -
          ```
          class X {
              ... whatever you want
          class Y {
              ... whatever you want
          };
          try {
              bunch of statements
              somewhere in here, or in the functions that are called:
                  throw X();
                  or
                  throw Y();
              }
          catch (X& x) {
              do something with an X exception
              }
          catch (Y& y) {
              do something with a Y exception
              }
          ... continue processing
              e.g. try again
          ```
        - *all of the catches are ignored unless there is a matching throw*
        - *when catch X is finished, skips over catch Y*
- **Can catch in more than one place**

- **Can catch, throw something else, rethrow the same exception**

  -
    ```
    class X {
            ... whatever you want
    class Y {
            ... whatever you want
    };
    try {
            bunch of statements
            try {
                    bunch of statements

                    somewhere in here, or in the functions that are called:
                            throw X();
                    somewhere in here, or in the functions that are called:
                            throw Y();
                    }
                    catch (X& x) {
                            do something with an X exception
                            throw;          // rethrow the same exception
                            or
                            throw Y();     // throw a different exception
                            }
            somewhere in here, or in the functions that are called:
                    throw X();
                    or
                    throw Y();
            }
    catch (X& x) {
            do something with an X exception
            }
    catch (Y& y) {
            do something with a Y exception
            }
    ... continue processing
            e.g. try again
    ```

- **What happens with uncaught exception?**
  - **if nobody catches it, there is a default catcher hidden in the run-time environment that catches everything and terminates the program**
  - **you can catch all exceptions with**
    - catch (...) { // three dots
    - cout << "some kind of exception caught" << endl;
    - }
- **In standard C++, memory allocation failure can be caught like this:**
  - **catch the bad_alloc exception**

    -
      ```
      #include <new>
      try {
              code that might allocate too much memory
              }
      catch (bad_alloc& x) {
      ```

```
                cout << "memory allocation failure" << endl;
                // do whatever you want
                }
```

- **Lots of other possibilities**
    - **can have a class hierarchy of exception types**
        - *catch by base class type, etc*
    - **can catch & rethrow exceptions**
    - **Standard library has some standard exception types that are thrown**
        - *e.g. bad_alloc*
    - **basic idea is easy to use!**
- **Only one thing to watch out for:**
    - **memory leaks while unwinding the stack**
        -
          ```
          class Thing {
          public:
                  Thing ();
                  ~Thing() {does something}
          };
          void foo ()
          {
                  Thing t;
                  Thing * t_ptr;
                  t_ptr = new Thing;
                  goo();
                  ...
          }
          ```
    - **Problem:**
        - *if goo throws an exception, foo is forced to return from the point of the call.*
        - *normal action on a return is to run the destructor function on local variables.*
            - t is a Thing, so its destructor ~Thing() is run
            - t_ptr is a pointer to Thing - it is popped off the stack, but because POINTERS ARE A BUILT-IN TYPE (like int) t_ptr doesn't have distructor, so the memory it is pointing to won't get deallocated. - can have a memory leak
    - **Fixes**
        - *catch all exceptions in foo and deallocate as needed*
        - *better - put such pointers inside an object with a dtor - "managed pointer" - can make them safer, better - later*