Topic 3 (Part IV: I/O Libraries and Exception Handling)

C++ advanced features review: *when can/should I use them?*
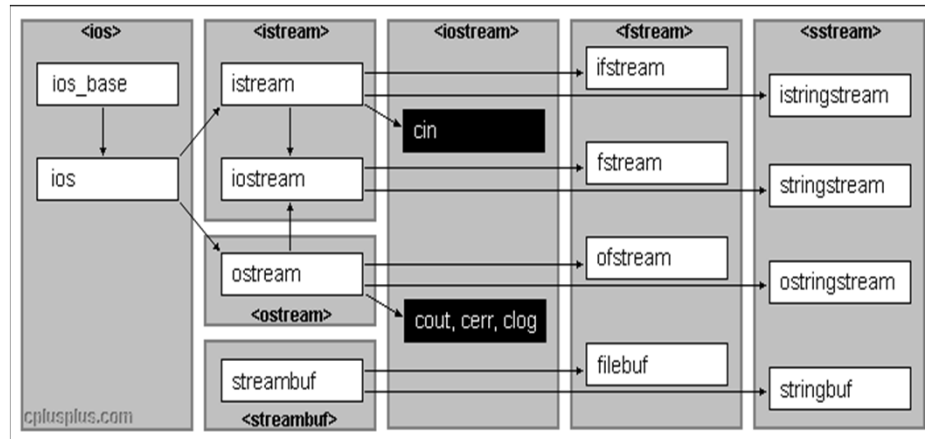
資料結構與程式設計

Data Structure and Programming

Sep, 2012

Outline

◆ C++ standard I/O
- Introduction
- Class hierarchy and included files
- Class data members and member functions
- I/O manipulators
- Tying istream to ostream

◆ File I/O

◆ String stream

1

Key Concept #1: C++ Stream Classes

<ios>	<istream>	<iostream>	<fstream>	<sstream>
ios_base	istream		ifstream	istringstream
		cin		
ios	iostream		fstream	stringstream
	ostream	ofstream		ostringstream
	<ostream>	cout, cerr, clog		
	streambuf	filebuf	stringbuf
cplusplus.com	<streambuf>			

For more information, recommended:
http://www.cplusplus.com/reference/iostream/

Stream classes, objects, and manipulators

◆ "Stream", a nice name
  ➔ Data are conveyed in a steam by "<<" or ">>"
1. Header files
   ● iostream, fstream, sstream, iomanip
2. Classes
   ● istream, ostream, iostream, ifstream, ofstream, fstream, istringstream, ostringstream
3. Objects
   ● Standard: cin, cout, cerr, clog
   ● User defined
4. Manipulators
   ● dec, endl, ends, flush, hex, oct, left, right, ws, setbase(n), setw(n), setioflags(i), resetioflags(i), setfill(c), setprecision(n)
5. Member functions

C++ Standard I/O Library Files

◆ `<iostream>`
- Basic services for ALL stream-I/O operations
- Defines `cin`, `cout`, `cerr` and `clog`
- For both unformatted- and formatted-I/O services

◆ `<iomanip>`
- Formatted I/O with parameterized stream manipulators

◆ `<fstream>`
- User-controlled file processing

◆ `<sstream>`
- String manipulations as I/O stream

Key Concept #2:
Standard I/O Stream Objects

Standard Input
◆ `cin`
- Connected to the standard input device, usually the keyboard

Standard Output
◆ `cout`
- Connected to the standard output device, usually the display screen

◆ `cerr`
- Connected to the standard error device
- Unbuffered - output appears immediately

◆ `clog`
- Connected to the standard error device
- Buffered - output is held until the buffer is filled or flushed

Key Concept #3:
User Defined Stream Objects

◆ File I/O
```
ifstream inFile("test.in");
ofstream outFile("test.out");
fstream ioFile;
if (!inFile) {
    cerr << "Cannot open file" << endl;
    exit(0);
}
int i, j, k;
inFile >> i >> j >> k;
outFile.close();
ioFile.open("test.io");
```

## Key Concept #4: File Stream

◆ A file is viewed by C++ as a sequence of bytes
◆ Ends either with an end-of-file marker (Ctrl-d for Linux and Ctrl-z for Windows) or at a system-recorded byte number (Why diff?)
◆ Communication between a program and a file is performed through stream objects
  ● <fstream> header file
    ■ Stream class templates
      • basic_ifstream – for file input
      • basic_ofstream – for file output
      • basic_fstream – for file input and output
    ■ Files are opened by creating objects of stream template specializations
      • (i/o)fstream are the char-type template specializations

# (FYI) basic_iostream

◆ Actually, in <iostream>, the I/O stream classes are defined as basic_iostream template classes

- template <class Elem, class Tr = char_traits<Elem> >
  class basic_iostream : public basic_istream<Elem, Tr>,
                                    public basic_ostream<Elem, Tr>

  { ... };

---
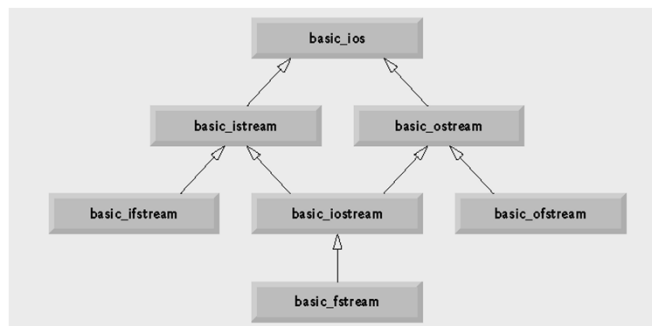
# (FYI) iostream vs. basic_iostream

◆ istream

- typedef basic_iostream<char, char_traits<char> > iostream;
- Represents a specialization of basic_istream
- Enables char input

◆ ostream

- Represents a specialization of basic_ostream
- Enables char output

◆ iostream

- Represents a specialization of basic_iostream
- Enables char input and output

# Key Concept #5: Open a file

◆ Two methods
  ● By passing arguments to (i/o)fstream constructor
  ● By calling member function open()
◆ Two arguments
  ● A filename  // mandatory; char*, not string
  ● A file-open mode  // optional; default = "out" for ostream, "in" for istream
    ■ Can use '|' for multiple modes
    ■ fstream fstr("test.txt", fstream::in | fstream::out | fstream::app);

| Mode | Description |
|------|-------------|
| ios::app | Append all output to the end of the file. |
| ios::ate | Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file. |
| ios::in | Open a file for input. |
| ios::out | Open a file for output. |
| ios::trunc | Discard the file's contents if they exist (this also is the default action for ios::out). |
| ios::binary | Open a file for binary (i.e., nontext) input or output. |

---

# fstream object

◆ open(): takes the same arguments as constructor
◆ Note: you cannot "copy" a stream object
  ● So, vector<ifstream> is not possible
◆ bool operator !()
  ● Returns true if either the failbit or badbit is set
  ● if (!fin) {
        cerr << "Open file failed..." << endl;
        exit(-1);
    }

# Key Concept #6: Close a file

◆ Releases the file resource (recommended!!!)
◆ Two methods
   - By destructor (exit the scope)
   - By calling member function close()

# Key Concept #7: I/O Stream Manipulators

1. endl
2. Number base (sticky)
   - hex (e.g. 0x38ab), oct (e.g. 0236), dec (all others)
   - showbase(), setbase(int)    // int = 16, 8, 10
3. Precision of floating-point numbers (sticky)
   - fixed, scientific
   - setprecision(int)
   - Note: precision(int) is a member function
4. Field width (not sticky)
   - setw(int)  // c.f. "width()" member function
   - For both istream (input size) and ostream (display size)

# I/O Stream Manipulators

5. Alignment (sticky)
   - left, right
   - internal (padding fill characters between sign and magnitude)
6. I/O formatting (sticky)
   - showpoint, noshowpoint
   - showpos, noshowpos
   - uppercase, nouppercase
   - boolalpha, noboolalpha
   - setfill (cf. fill() member function)
   - skipws
7. Flush stream buffer
   - flush

# Sticky or not sticky?

◆ Most IO manipulators are "sticky"
   - Exception: field width
◆ "Sticky" to the manipulated object
   - Not across to another object of the same stream class, or any other object of other stream classes

## Use of Manipulators

```
int main()
{
   int i = 100;
   fstream iof("ttt");
   if (!iof) { cerr << "Error" << endl; exit(0); }
   iof << hex << i << endl;
   iof.close();

   int j;
   iof.open("ttt");
   iof >> j;

   cout << setw(10) << right << j << endl;
}
// What's in file "ttt"? What's the output??
```

What's the difference?

```
int main()
{
   int i = 100;
   fstream iof("ttt");
   if (!iof) { cerr << "Error" << endl; exit(0); }
   iof << hex << i << endl;
   iof.close();

   int j;
   iof.open("ttt");
   iof >> dec >> j;

   cout << setw(10) << right << j << endl;
}
// What's in file "ttt"? What's the output??
```

(FYI) About I/O Manipulators

◆ About floating number display
- fixed --- in fixed-point notation (e.g. 3.14159)
- scientific --- in scientific notation (e.g. 3.14159e+002)
- (none) --- in default floating-point notation; floating-point number's value determines the output format

◆ About the precision of display
- setprecision(*numDigits*)
  - For "fixed" and "scientific", *numDigits* is the number of digits after the decimal point
  - For default floating-point notation, *numDigits* is the total number of digits to display

---

Key Concept #7:
User-Defined Stream Manipulators

◆ Programmers can create their own stream manipulators
- cf: ostream member function:
  ostream& operator << (ostream& (*p)(ostream&));

◆ [e.g.] Output stream manipulators
- Must have <u>return type</u> and <u>parameter type</u> as `ostream &`
- e.g.

```
ostream& myEndl (ostream& os) {
    return (os << endl << "Ric> ");
}
==============================
int main() {
    cout << myEndl;
}
```

Key Concept #8:
Formatted vs. Unformatted I/O

- ◆ Formatted I/O
  - ● "High-level", bytes are grouped into meaningful units
    - ■ Integers, floating-point numbers, characters, etc.
    - ■ Satisfactory for most I/O other than high-volume file processing
  - ● I/O operations are sensitive to data types
    - ■ Improper data cannot "sneak" through
  - ● Using operators "<<" and ">>", I/O manipulators
- ◆ Unformatted I/O
  - ● Low-level, individual bytes are the items of interest
  - ● High-speed, high-volume
  - ● Not particularly convenient for programmers
  - ● Member functions (e.g. get, getline, put, read, write...)
  - ● May have portability problem

Type-Safe I/O (Formatted I/O)

- ◆ << and >> operators are overloaded to accept data of specific types
  - ● Attempts to input or output a user-defined type that << and >> have not been overloaded will result in compiler errors
- ◆ If unexpected data is processed, error bits are set
  - ● User may test the error bits to determine I/O operation success or failure
- ◆ ostream& operator <<
  - ● Does not print out until '\n" or "flush()" is called
- ◆ istream& operator >>
  - ● Stop at white space, but not process until '\n' is entered

Key Concept #9: Overloading "<<" operator for user-defined types

```
class MyClass {
   // friend to "global domain"
   //         so that it can be accessed by "<<(cout, m)"
   friend ostream& operator <<
         (ostream& os, const MyClass& m);
}; // Why friend? friend to whom???

ostream& operator << (ostream& os, const MyClass& m) {
   os << m._dataMember1 << " is "  << m._dataMember2...
   return os; // Why return "os"? Return to whom?
}

int main()
{
   MyClass m(100);
   cout << m << endl;
}
```

"friend" is NOT a must, just a custom

```
class MyClass {


   ostream& operator <<                Not a member function!!
      (ostream& os, const MyClass& m);
};

ostream& operator << (ostream& os, const MyClass& m) {
   os << m.getData1() << " is "  << m.getData2()...
   return os;
}

int main()
{
   MyClass m(100);
   cout << m << endl;
}
```

However, it is a good practice to add a friend entry in the class so that users can be aware of that such overloading exists.

Try this...

◆
```
int i;
while (true) {
    cin >> i;
    // ... do something on i,
    // for example:
    cout << i << endl;
}
```

➔ What will you see if we enter the char 'a'?

Key Concept #10: I/O Stream State Bits

◆ Control the state of the stream (as ios data members)
  - failbit
    - ■ Set if input data is of wrong type (format error)
    - ■ <u>Data still remains in stream buffer</u>
    - ■ Usually can be recovered
  - badbit
    - ■ Set if stream extraction operation fails (more serious)
    - ■ Usually difficult to recover
  - eofbit
    - ■ Set if the end of file is reached during stream input
  - goodbit
    - ■ ! (failbit | badbit | eofbit)

I/O Stream State Bits

◆ Functions
- bool eof() const;
  - ■ Returns `true` when end-of-file has occurred
  - ■ [What's wrong??] while (!infile.eof()) { infile >> ch; ... }
- good(), fail(), bad()
- rdstate()
- clear(iostate state=ios::goodbit)
  - ■ Sets the specified bit for the stream
  - ■ Default argument is `goodbit`
  - ■ Examples
    `cin.clear();`
    - • Clears `cin` and sets `goodbit`
    `cin.clear( ios::failbit );`
    - • Sets `failbit`
- setstate(iostate state) ➔ clear(rdstate() | state)

To fix the previous problem...

◆
```
int i;
while (true) {
   cin >> i;
   while (cin.fail()) {
      cin.clear();
      string str;
      cin >> str;
      cerr << "Error: " << str
           << " is NOT an int!!" << endl;
      cin >> i;
   }
   cout << i << endl;
}
```

14

while (fstream)?

◆ What does this do?

```
int main()
{
    ifstream inf("aaa.txt");
    char ch;
    while (inf >> ch) cout << ch;
}
```

◆ ios::operator void* ( ) const
  ● Converted to void*; return NULL if failbit or badbit is set

Key Concept #11:
Flags for I/O Stream Printing Format

◆ Member function flags()
  ● With no argument
    ■ Returns a value of type fmtflags
      • Represents the current format settings
  ● With a fmtflags as an argument
    ■ Sets the format settings as specified
    ■ Returns the prior state settings as a fmtflags
  ● Initial return value may differ across platforms
  ● Type fmtflags is of class ios_base

What will be the output?

```
int main() {
   int i = 100;
   ofstream outf("ttt");
   outf << showbase << hex << i << endl;
   ios_base::fmtflags origFlags = outf.flags();
   outf.close();

   ifstream inf("ttt");
   inf.flags(origFlags);
   inf >> i;
   cout << setw(10) << right << i << endl;
}
```
◆What's the content in "ttt"?

Key Concept #12: Unformatted I/O

◆ Think: sometimes you just want to read/write a file as a "stream of bytes"
  - You don't care/know about the type of each piece of data
  - → Unformatted I/O
◆ Use member functions to do file accesses

istream::get

1. With no arguments
   - int get ();
   - Returns one character input from the stream
     - Any character, including white-space and non-graphic characters
   - Returns EOF when end-of-file is encountered
2. With a character-reference argument
   - istream& get (char& c);
   - Stores input character in the character-reference argument
   - Returns a reference to the istream object

istream::get

3. With three arguments: a character array, a size limit and a delimiter (default delimiter is ' \n')
   - istream& get (char* s, streamsize n);
     istream& get (char* s, streamsize n, char delim);
     istream& get (streambuf& sb);
     istream& get (streambuf& sb, char delim);
   - Reads and stores characters in the character array
   - Terminates at one fewer characters than the size limit or upon reading the delimiter
     - Delimiter is left in the stream, not placed in array
   - Null character is inserted after end of input in array
   - [note] "streamsize" may be platform dependent, usually signed int or signed long.

istream::getline

◆ istream& getline (char* s, streamsize n);
istream& getline (char* s, streamsize n, char delim);
- Similar to the three-argument version of `get`
  - Except the delimiter *is* removed from the stream
- Three arguments: a character array, a size limit and a delimiter (default delimiter is ' \n')
  - Reads and stores characters in the character array
  - Terminates at one fewer characters than the size limit or upon reading the delimiter
    - Delimiter is removed from the stream, but not placed in the array
  - Null character is inserted after end of input in array

ostream::put

◆ ostream& put (char c);   // unformatted
- Outputs a character
- Returns a reference to the same `ostream` object
  - Can be cascaded
- Can be called with a numeric expression that represents an ASCII value
- Examples
  - `cout.put( 'A' );`
  - `cout.put( 'A' ).put( '\n' );`
  - `cout.put( 65 );`

More Unformatted I/O Functions

◆ istream& read (char* s, streamsize n);
  - Inputs some number of bytes to a character array
  - If fewer characters are read than the designated number, fail bit is set

◆ streamsize gcount ( ) const;
  - Reports number of characters read by last input operation

◆ ostream& write (const char* s , streamsize n);
  - Outputs some number of bytes from a character array

More istream member functions

◆ istream& ignore
  (streamsize n = 1, int delim = EOF);
  - Reads and discards a designated number of characters or terminates upon encountering a designated delimiter

◆ istream& putback (char c);
  - Places previous character obtained by a get from the input stream back into the stream

◆ int peek ( );
  - Returns the next character in the input stream, but does not remove it from the stream

Key Concept #13:
Tying an Input Stream to an Output Stream

◆ istream member function tie
  ● ostream* tie ( ) const;
    ■ Returns a pointer to the tied output stream
  ● ostream* tie (ostream* tiestr);
    ■ Ties the istream object to *tiestr* and returns a pointer to the ostream object previously tied
◆ Synchronizes an istream and an ostream
  ● Ensures outputs appear before their subsequent inputs
◆ By default, the standard objects cin, cerr and clog are tied to cout (Why?)
  ● Examples
    ■ cin.tie(&cout);
      • Ties standard input to standard output
      • C++ performs this operation automatically
    ■ inputStream.tie(0);
      • Unties inputStream from the ostream it is tied to

---

istream::tie Example

```
int main () {
  ostream *prevstr;
  ofstream ofs;
  ofs.open ("test.txt");
  cout << "tie example:" << endl;
  *(cin.tie()) << "This is inserted into cout";
  prevstr = cin.tie(&ofs);
  *(cin.tie()) << "This is inserted into the file";
  cin.tie (prevstr);
  ofs.close(); return 0;
}
```

Key Concept #14: File-position pointer

◆ The **byte number** of the next byte to be read or written

◆ `seekg()` for ifstream   and
  `seekp()` for ofstream
  - Repositions the file-position pointer to the specified location
  - Two prototypes
    - seekg(pos) or seekg(offset, direction)

◆ `tellg()` for ifstream   and
  `tellp()` for ofstream
  - Returns current position of the file-position pointer as type long

Seek direction

◆ `ios::beg` – default, position relative to the beginning
  `ios::cur` – relative to current position
  `ios::end` – relative to the end

◆ `Examples`
  - `fileObject.seekg( n );`
    - Position to the *n*th byte of `fileObject`
  - `fileObject.seekg( n, ios::cur );`
    - Position *n* bytes forward in `fileobject`
  - `fileObject.seekg( n, ios::end );`
    - Position *n* bytes back from end of `fileObject`
  - `fileObject.seekg( 0, ios::end );`
    - Position at end of `fileObject`

Key Concept #15: Random-Access Files

◆ Necessary for instant-access applications
  - Such as transaction-processing systems
    - cf: use ">>", "<<" for *sequential file* access
  - A record can be inserted, deleted or modified without affecting other records
◆ Various techniques can be used
  - Require that all records be of the same length, arranged in the order of the record keys
    - Program can calculate the exact location of any record
      - Base on the record size and record key
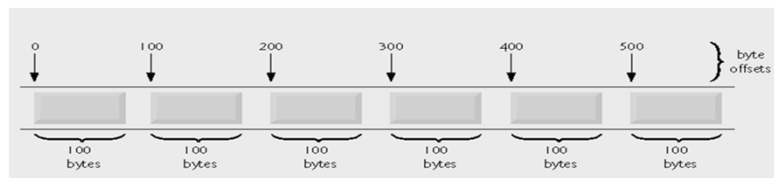
Functions for Random-Access Files

◆ istream& read(char *str, streamsize nBytes)
  - Read a number of bytes from the current file position in the stream into an object
◆ ostream& write
  (const char *str, streamsize nBytes)
  - Writes a number of bytes from a location in memory to the stream

Random-Access Files

```
int main() {
  StudentRecord rec;
  ofstream outf("studentRecord.dat");
  for (int i = 0; i < numRecords; ++i) {
    cin >> rec;
    outf.write(reinterpret_cast<const char *>(&rec),
               sizeof(StudentRecord));
  }
  outf.close();
  ifstream inf("studentRecord.txt");
  for (int i = 0; i < numRecords; ++i)
    inf.read(reinterpret_cast<const char *>(&rec),
             sizeof(StudentRecords));
}
```

Operator `reinterpret_cast`

StudentRecord rec;
outf.write(reinterpret_cast<const char *>(&rec),
sizeof(StudentRecord));

➔

◆ Casts a pointer of one type to an unrelated type
  ● Also converts between pointer and integer types
◆ Is performed at compile time
  ● Does not change the value of the object pointed to
◆ Could lead to serious execution-time errors

23

Key Concept #16:
String Stream (Stream of string)

```
// #include <sstream>
int main()
{
    int i;
    cin >> i;

    ostringstream  st;
    st << i << " square is " << i * i;

    string str = st.str();

    cout << str << endl;
}
// What's the output??
```

String Stream

```
int main()
{
    int i;
    cin >> i;

    ostringstream  st;
    st << i << " square is " << i * i;
    string str = st.str();
    cout << str << endl;


    st << i << " is " << i;
    str = st.str();
    cout << str << end
}
// What's the output??
// How to clear the previous string? clear()?
```

The Solution is.... ^^|||

```
int main()
{
   int i;
   cin >> i;

   ostringstream  st;
   st << i << " square is " << i * i;
   string str = st.str();
   cout << str << endl;

   st.str("");
   st << i << " is " << i;
   str = st.str();
   cout << str << end
}
```

Key Concept #17:
Exceptions in your program

1. Runtime error (system exception)
   - Segmentation fault, bus error, etc
   - e.g. class bad_alloc
2. User-defined exception
   - If that happens, I don't want to handle it...
3. Interrupt
   - Control-C, etc

What is exception handling?

◆ When an exception happens, detect it and stop the program "gracefully" (usually by going back to a command prompt), instead of terminating the program abruptly

◆ Purposes
- To keep the partial results
- Continue the program without losing previous efforts
- (e.g. Windows crashing vs. "a program terminating abnormally")

Key Concept #18:
C++ Exception Handling Mechanism

◆
```
Try {
   // your main program
   // if (exception happens)
   //    throw exception!!
}
catch (ExpectedException1) {
   // exception handling code 1
}
// More exceptions to catch here
catch (...) {
   // The rest of the exceptions
   // note: "..." above is a reserved
   // symbol here
}
```

Predefined Classes for Exception Handling

◆ int, char*, etc
```
try {
    buf = new char[512];
    if( buf == 0 )
        throw "Memory alloc failure!";
} catch(const char * str ) {
    cout << "Exception raised: "
        << str << '\n';
}
```
◆ Predefined classes
```
try {
    ...
} catch (bad_alloc) {  // defined in "std"
    // Handle memory out
    ...
}
```

User-defined Classes for Exception Handling

◆
```
class MyException      {
    int  _type;
public:
    MyException(int i) : _type(i) {}
    void print() const { ... }
};
===============
int main()       {
    try {
        ...
        if (...) throw MyException(type);
    }
    catch (MyException& ex) {
        ex.print();
        // handle the exception here...
    }
}
```

27

Hierarchical exception handling

```
void g() {                      main() {
   ......                          try {
   if (.....)                         f();
      throw Ex4f();               } catch (Ex4F& e){
}                                    ......
void f() {                       } catch (Ex4Main& e) {
   try {                             ......
      .....                       } catch (...) {
      g();                           ...
   } catch (Ex4F& e4f) {          }
      // do something....      }
      if (.....)
         throw Ex4Main();
      else
         throw e4f;
   }
}
```

Key Concept #19: Limited Throw

◆ Limit only certain types of exceptions to pass through
  ● void f() throw(OnlyException&);
    ➔ Only exceptions of class "OnlyException" are allowed
  ● void f() throw();
    ➔ None of the exception is allowed
◆ If disallowed exception is thrown
  > terminate called after throwing an instance of 'xxxx'
  > Abort
  Uh??? Why do we limit the types of thow?
  ➔ catch the problematic code earlier!!

28

The Challenges in Exception Handling

1. Make sure the program is reset to a "clean state"

   - Memory used by unfinished routines needs to be released back to OS
   - All the data fields (e.g. _flags) needs to be made consistent

2. Be able to continue the execution

   - The unaffected data should not be deleted

Key Concept #20: Handling Interrupt

◆ Associate an interrupt signal to a handler
  - void signal(int sigNum, void sigHandler(sigNum));
◆ system-defined sigNum
  - SIGABRT   Abnormal termination
  - SIGFPE   Floating-point error
  - SIGILL   Illegal instruction
  - SIGINT   CTRL+C signal
  - SIGSEGV   Illegal storage access
  - SIGTERM   Termination request
◆ sigHandler()
  - Predefined: SIG_IGN(), SIG_DFL()
  - User-defined functions
    - void myHandler(int)

Interrupt Handler Implementation

1. Define interrupt handler function
   - Using "signal(int signum, sighandler_t handler)";
   → "signum" example: SIGINT → Control-C
   → "handler" is: "void myHandler(int)"
   → man signal
2. Define a flag to denote the detection of the interrupt
3. Initialize the flag to "undetected" (e.g. false)
4. Associate the target interrupt to the handler function
5. (In handler function)
   - Re-associate this target interrupt to "SIG_IGN" (why?)
   - Set the flag to "detected"
6. (In upper-level/main function)
   - Check the flag "detected"
   - If (detected) → reset to "undetected"; re-associate this interrupt with your handler;
   - Do something;

Random Number Generator by Control-C

```
static bool
    _ctrlCDetected = false;
unsigned rnGen(unsigned max)
{
  unsigned count = 0;
  while (1) {
    if (isCtrlCDetected()) {
      setCtrlCDectected(false);
      return count;
    }
    if (++count == max)
      count = 0;
  }
  return 0;
}
```

```
static void ctrlCHandler(int) {
  signal(SIGINT, SIG_IGN);
  if (!isCtrlCDetected())
    setCtrlCDectected(true);
}
static void ctrlCHandlerReset() {
  setCtrlCDectected(false);
  signal(SIGINT, ctrlCHandler);
}
int main(int argc, char** argv) {
  ctrlCHandlerReset();
  unsigned max = atoi(argv[1]);
  cout << rnGen(max) << endl;
  return 0;
}
```