Imperial College London

Advanced Programming

C++ Errors, Debugging & Exceptions

Adriana Paluszny

[Many of the slides are adapted from the original Bjarne Stroustrup slides on C++]

Your Program

1. Should produce the desired results for all legal inputs
2. Should give reasonable error messages for illegal inputs
3. Need not worry about misbehaving hardware
4. Need not worry about misbehaving system software
5. Is allowed to terminate after finding an error

3, 4, and 5 are true for beginner's code; often, we have to worry about those in real software.

Imperial College London

Sources of errors

- Poor specification
  - “What’s this supposed to do?”
- Incomplete programs
  - “but I’ll not get around to doing that until tomorrow”
- Unexpected arguments
  - “but `sqrt()` isn’t supposed to be called with `-1` as its argument”
- Unexpected input
  - “but the user was supposed to input an integer”
- Code that simply doesn’t do what it was supposed to do
  - “so fix it!”

Imperial College London

Kinds of Errors

- Compile-time errors
  - Syntax errors
  - Type errors
- Link-time errors
- Run-time errors
  - Detected by computer (crash)
  - Detected by library (exceptions)
  - Detected by user code
- Logic errors
  - Detected by programmer (code runs, but produces incorrect output)

Imperial College
London

Bad function arguments

- The compiler helps:
  - Number and types of arguments must match

```
int area(int length, int width)
{
   return length*width;
}

int x1 = area(7);             // error: wrong number of arguments
int x2 = area("seven", 2);  // error: 1st argument has a wrong type
int x3 = area(7, 10);         // ok
int x5 = area(7.5, 10);       // ok, but dangerous: 7.5 truncated to 7;
                              //      most compilers will warn you
int x = area(10, -7);         // this is a difficult case:
                              // the types are correct,
                              // but the values make no sense
```

Imperial College
London

Bad Function Arguments

- So, how about `int x = area(10, -7);` ?
- Alternatives
  - Just don’t do that
    - Rarely a satisfactory answer
  - The caller should check
    - Hard to do systematically
  - The function should check
    - Return an “error value” (not general, problematic)
    - Set an error status indicator (not general, problematic – don’t do this)
    - Throw an exception
- Note: sometimes we can’t change a function that handles errors in a way we do not like
  - Someone else wrote it and we can’t or don’t want to change their code

Imperial College London

Bad function arguments

- Why worry?
  - You want your programs to be correct
  - Typically, the writer of a function has no control over how it is called
    - Writing “do it this way” in the manual (or in comments) is no solution – many people don’t read manuals
  - The beginning of a function is often a good place to check
    - Before the computation gets complicated
- When to worry?
  - If it doesn’t make sense to test every function, test some

Imperial College London

How to report an error

- Return an "error value" (not general, problematic)

```
int area(int length, int width)    // return a negative value for bad input
{
   if(length <=0 || width <= 0) return -1;
   return length*width;
}
```

- So, "let the caller beware"

```
int z = area(x,y);
if (z<0) error("bad area computation");
// …
```

- Problems
  - What if I forget to check that return value?
  - For some functions there isn't a "bad value" to return (e.g., `max()`)

Imperial College
London

How to report an error

- Set an error status indicator (not general, problematic, don't!)

```
int errno = 0;     // used to indicate errors
int area(int length, int width)
{
  if (length<=0 || width<=0) errno = 7;   // the symbol || means or
  return length*width;
}
```

- So, "let the caller check"

```
int z = area(x,y);
if (errno==7) error("bad area computation");
// …
```

- Problems
  - What if I forget to check `errno`?
  - How do I pick a value for `errno` that's different from all others?
  - How do I deal with that error?

How to report an error

- Report an error by throwing an exception

```
class Bad_area { };     // a class is a user defined type
                        // Bad_area is a type to be used as an exception

int area(int length, int width)
{
  if (length<=0 || width<=0) throw Bad_area{};   // note the {} – a value
  return length*width;
}
```

- Catch and deal with the error (e.g., in **main()**)

```
try {
  int z = area(x,y);    //  if area() doesn’t throw an exception
}                       // make the assignment and proceed
catch(Bad_area) {  // if area() throws Bad_area{}, respond
  cerr << "oops! Bad area calculation – fix program\n";
}
```

Imperial College London

What is an exception?

- C++ provides a mechanism, called exception handling, to help deal with errors.
- The fundamental idea is to separate:
  - the detection of an error (which should be done in a called function)
  - from the handling of an error (which should be done in the calling function) while ensuring that a detected error cannot be ignored.

Imperial College
London

Exceptions

- Exception handling is general
  - You can’t forget about an exception: the program will terminate if someone doesn’t handle it (using a **try … catch**)
  - Just about every kind of error can be reported using exceptions
- You still have to figure out what to do about an exception (every exception thrown in your program)
  - Error handling is **never** really simple

Imperial College
London

Out of range

- Try this

```
vector<int> v(10);    // a vector of 10 ints,
                 // each initialized to the default value, 0,
                 // referred to as v[0] .. v[9]
for (int i = 0; i<v.size(); ++i) v[i] = i;  // set values
for (int i = 0; i<=10; ++i)          // print 10 values (???)
   cout << "v[" << i << "] == " << v[i] << endl;
```

- vector’s **operator[ ]** (subscript operator) – bad index will be reported in debug but not in release
- The default behavior can differ
- You can’t make this mistake with a range-**for**

Imperial College London

How does an exception work?

1) A function that throws an exception is called from within a try block
2) If the function throws an exception, the function terminates and the try block is immediately exited.
    1) A catch block to process the exception is searched for in the source code immediately following the try block.
    2) If a catch block is found that matches the exception thrown, it is executed.
    3) If no catch block that matches the exception is found, the program terminates.

Imperial College London

An exception can be of many types

```
int main( )
{
  try
  {
    double x;
    cout << "Enter a number: ";
    cin >> x;
    if (x < 0) throw "Bad argument!";
    cout << "Square root of " << x <<  " is " << sqrt(x);
  }
  catch(char *str)
  {
       cout << str;
  }
  return 0;
}
```

Imperial College
London

An exception can be of type int

```
try
   {
      quotient = divide(num1, num2);
       cout << "The quotient is " << quotient << endl;
   }
catch (int exceptionInt)
   {
      goodResult=false;
       cout <<“Error code”<<exceptionInt;
   }
```

```
in divide(..) you would have something like:
if (denominator == 0) throw 404;
```

Imperial College
London

Several exceptions

Multiple catch blocks can be attached to the same block of code. The catch blocks should handle exceptions of different types

```
try
{...}
catch(int iEx)
{...}
catch(char *strEx)
{...}
catch(double dEx)
{...}
```

Imperial College London

class Exceptions

- You can define “class exceptions”

- The catch block must be designed to catch an object of the exception class
- Exception class object can pass data to exception handler via data members
- An exception class may have:
  - no members: used only to signal an error
  - members: pass error data to **`catch`** block
- A class can have more than one exception class

- There are a number of ‘standard exception’ objects

Imperial College
London

Exceptions

- For now, just use exceptions to terminate programs gracefully, like this

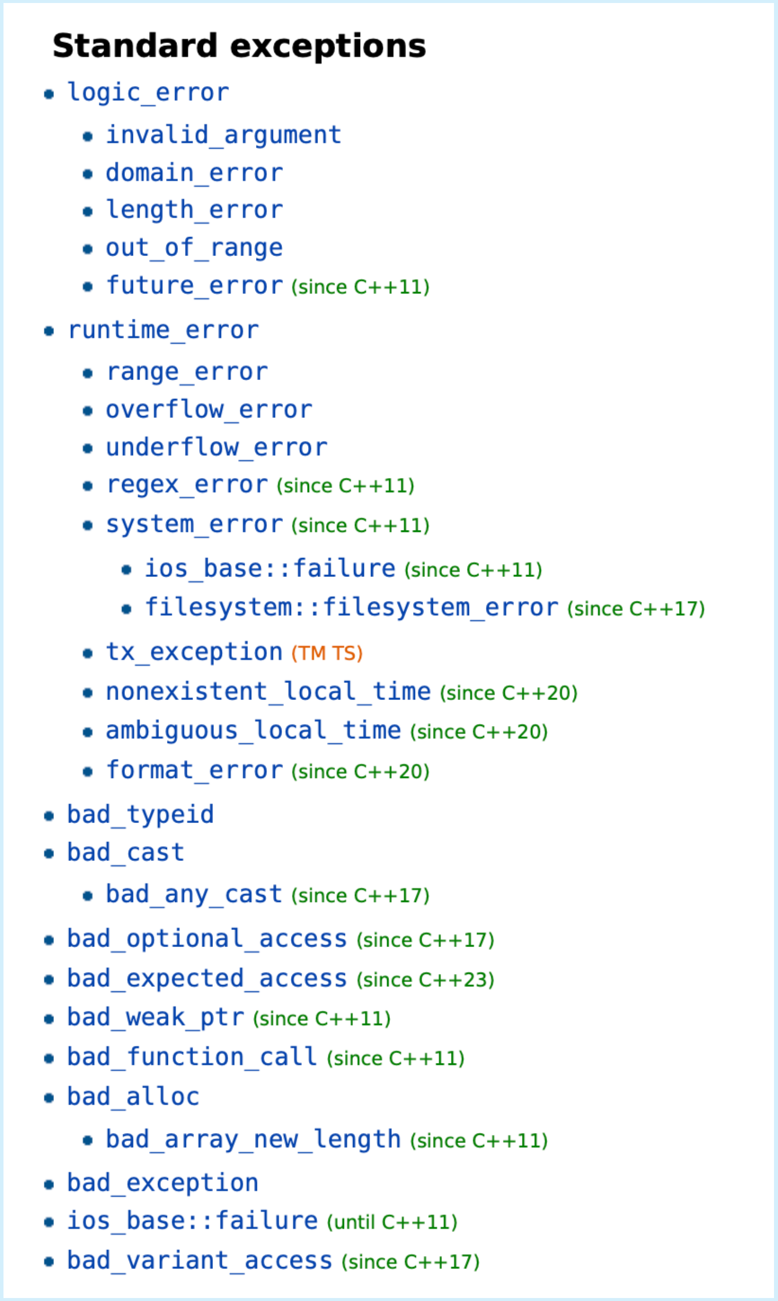
```
int main()
try
{
    // …
}
catch (out_of_range&) { // out_of_range exceptions
    cerr << "oops – some vector index out of range\n";
}
catch (…) {         // all other exceptions
    cerr << "oops – some exception\n";
}
```

Exceptions are advanced mechanisms to control the flow of errors in your programme.

Imperial College London

Exceptions

- There are a number of exception classes in the standard library

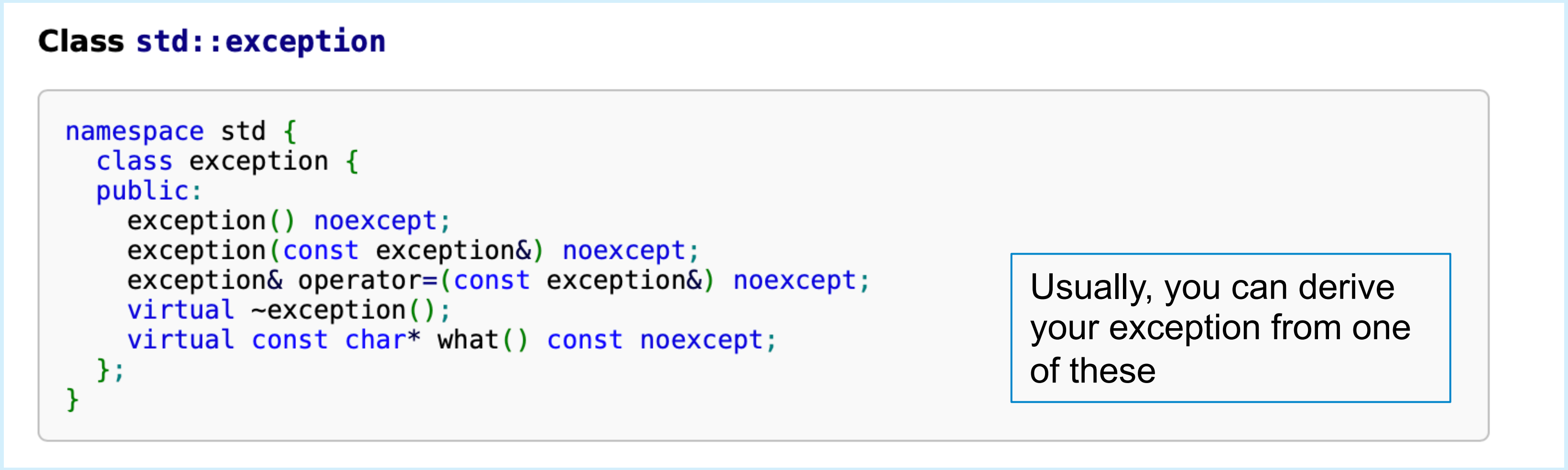


Standard exceptions

- `logic_error`
  - `invalid_argument`
  - `domain_error`
  - `length_error`
  - `out_of_range`
  - `future_error` (since C++11)
- `runtime_error`
  - `range_error`
  - `overflow_error`
  - `underflow_error`
  - `regex_error` (since C++11)
  - `system_error` (since C++11)
    - `ios_base::failure` (since C++11)
    - `filesystem::filesystem_error` (since C++17)
  - `tx_exception` (TM TS)
  - `nonexistent_local_time` (since C++20)
  - `ambiguous_local_time` (since C++20)
  - `format_error` (since C++20)
- `bad_typeid`
- `bad_cast`
  - `bad_any_cast` (since C++17)
- `bad_optional_access` (since C++17)
- `bad_expected_access` (since C++23)
- `bad_weak_ptr` (since C++11)
- `bad_function_call` (since C++11)
- `bad_alloc`
  - `bad_array_new_length` (since C++11)
- `bad_exception`
- `ios_base::failure` (until C++11)
- `bad_variant_access` (since C++17)

Imperial College
London

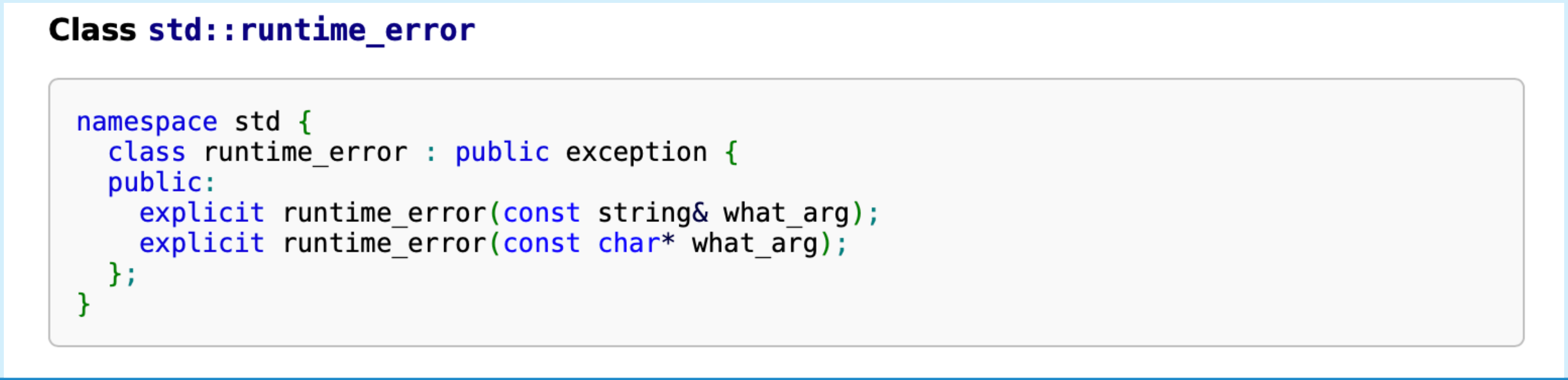
The base class for exceptions



Class `std::exception`

```
namespace std {
  class exception {
  public:
    exception() noexcept;
    exception(const exception&) noexcept;
    exception& operator=(const exception&) noexcept;
    virtual ~exception();
    virtual const char* what() const noexcept;
  };
}
```

Usually, you can derive your exception from one of these



Class `std::runtime_error`

```
namespace std {
  class runtime_error : public exception {
  public:
    explicit runtime_error(const string& what_arg);
    explicit runtime_error(const char* what_arg);
  };
}
```

Imperial College London

You can use exceptions to maintain flow

```
// Get the rectangle's length.
cout << "Enter the rectangle's length: ";
cin >> length;

// Store the length in the myRectangle object.
tryAgain = true;
while (tryAgain)
{
  try
  {
    myRectangle.setLength(length);
    // If no exception was thrown, then the
    // next statement will execute.
    tryAgain = false;
   }
   catch (Rectangle::NegativeLength)
   {
     cout << "Please enter a non-negative length: ";
     cin >> length;
   }
 }
```

Imperial College London

Exceptions

- To differentiate between a negative width and length, two classes can be created and each called based on the type of error
- In Rectangle.h include:

```
public:
  class NegativeWidth { }; // Exception class for a negative width
  class NegativeLength { }; // Exception class for a negative length
```

- Then catch these two new objects:

```
try
{
 myRectangle.setWidth(width);
 myRectangle.setLength(length);
 cout << "The area of the rectangle is ‟ << myRectangle.getArea() << endl;
}
catch (Rectangle::NegativeWidth)
{ cout << "Error: A negative value was given "<< "for the rectangle's width.\n"; }
catch (Rectangle::NegativeLength)
{ cout << "Error:A negative value was given "<< "for the rectangle's length.\n"; }
```

Uncaught exceptions

- An exception may be uncaught if
  - there is no **`catch`** block with a data type that matches the exception that was thrown, or
  - it was not thrown from within a **`try`** block
- The program will terminate in either case

EXERCISE

- Write a function that divides one number by another one `a/b`,
  - `double divide(double a, double b)`
- Write a main that loops over 10,000 pairs of random doubles (between 0 and 1), divides them and accumulates the division into a variable `total_div`.

Imperial College London

How to look for errors

- When you have written (drafted?) a program, it'll have errors (commonly called "bugs")
  - It'll do something, but not what you expected
  - How do you find out what it actually does?
  - How do you correct it?
  - This process is usually called "debugging"

Imperial College London

Debugging

- How ***not*** to do it

```
while (program doesn’t appear to work) { // pseudo code
   Randomly look at the program for something that “looks odd”
   Change it to “look better”
}
```

- Key question

  How would I know if the program actually worked correctly?

Imperial College London

Program structure [Recap]

- Make the program easy to read so that you have a chance of spotting the bugs
  - Comment
    - Explain design ideas
  - Use meaningful names
  - Indent
    - Use a consistent layout
    - Your IDE tries to help (but it can't do everything)
      - You are the one responsible
  - Break code into small functions
    - Try to avoid functions longer than a page
  - Avoid complicated code sequences
    - Try to avoid nested loops, nested if-statements, etc.
      (But, obviously, you sometimes need those)
  - Use library facilities

Imperial College
London

First get the program to compile [Recap]

- Is every string literal terminated?

```
cout << "Hello, << name << '\n';   // oops!
```

- Is every character literal terminated?

```
cout << "Hello, "  << name << '\n;    // oops!
```

- Is every block terminated?

```
if (a>0) { /* do something */
 else { /* do something else */ }  // oops!
```

- Is every set of parentheses matched?

```
if (a      // oops!
     x = f(y);
```

- The compiler generally reports this kind of error “late”
  - It doesn’t know you didn’t mean to close “it” later

Do not let errors accumulate!

They will occlude each other and are much harder to find if nested

Always deal with the first error first.

Imperial College
London

First get the program to compile

- Is every name declared?
  - Did you include needed headers?

- Is every name declared before it's used?
  - Did you spell all names correctly?

```
int count; /* … */ ++Count;          // oops!
char ch;        /* … */ Cin>>c;      // double oops!
```

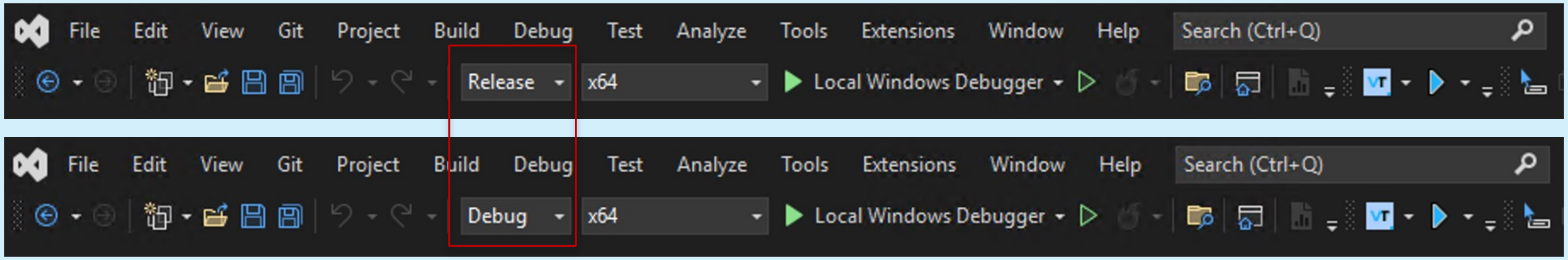
- Did you terminate each expression statement with a semicolon?

```
x = sqrt(y)+2// oops!
z = x+3;
```

Imperial College London

Compilation Configurations

- **RELEASE** mode:
  - Release usually has optimizations enabled
  - 10-20 times faster
  - No debug
- **DEBUG** mode
  - In debug configuration, your program compiles with full symbolic debug information (Symbol files)
  - No optimization.

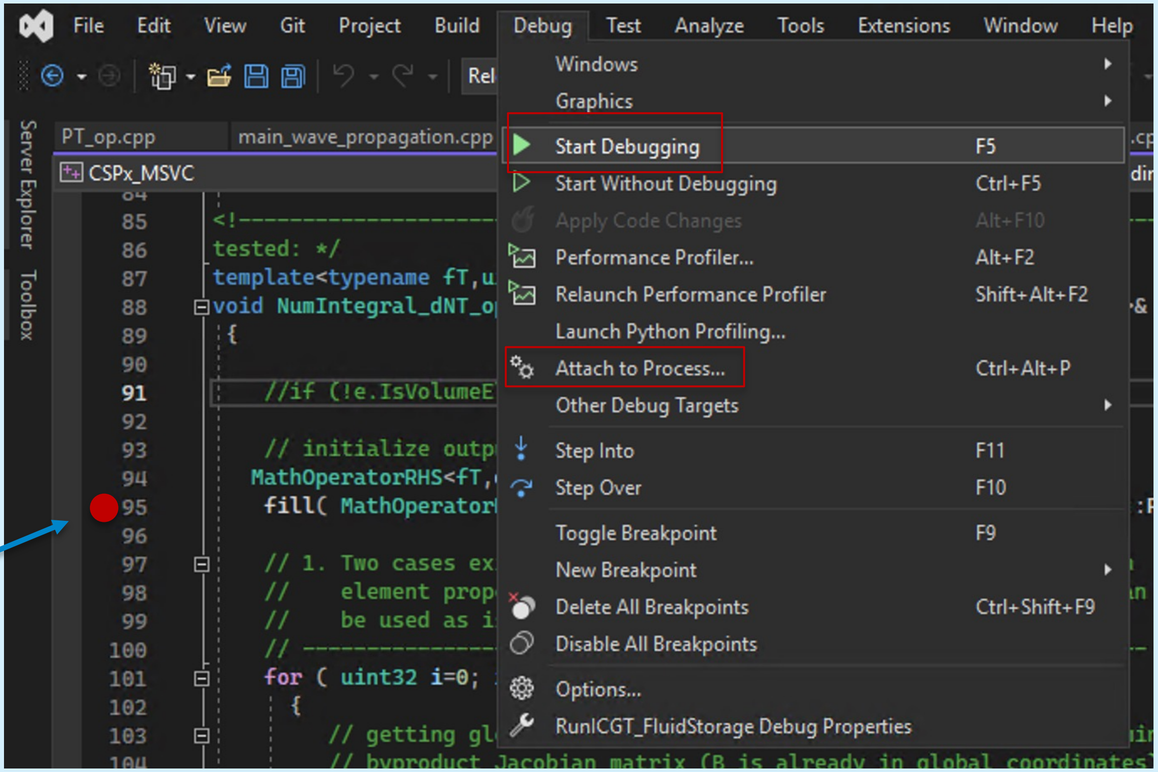


Imperial College
London

Using Debugging

- Step 1: Compile with DEBUG flag
- Step 2: Run your code in DEBUG mode (or attach to it)

Break point



File
Edit
View
Git
Project
Build
Debug
Test
Analyze
Tools
Extensions
Window
Help
Server Explorer
Toolbox
PT_op.cpp
main_wave_propagation.cpp
CSPx_MSVC
Windows
Graphics
Start Debugging F5
Start Without Debugging Ctrl+F5
Apply Code Changes Alt+F10
Performance Profiler... Alt+F2
Relaunch Performance Profiler Shift+Alt+F2
Launch Python Profiling...
Attach to Process... Ctrl+Alt+P
Other Debug Targets
Step Into F11
Step Over F10
Toggle Breakpoint F9
New Breakpoint
Delete All Breakpoints Ctrl+Shift+F9
Disable All Breakpoints
Options...
RunICGT_FluidStorage Debug Properties

```
<!---------------------
tested: */
template<typename fT,u
void NumIntegral_dNT_o
{

    //if (!e.IsVolumeE

    // initialize outp
   MathOperatorRHS<fT,
    fill( MathOperator

    // 1. Two cases ex
    //    element prop
    //    be used as i
    // ---------------
    for ( uint32 i=0;
    {
        // getting gl
        // byproduct Jacobian matrix (B is already in global coordinates)
```

Imperial College
London

Debugging

- Carefully follow the program through the specified sequence of steps
  - Pretend you’re the computer executing the program
  - Does the output match your expectations?
  - If there isn’t enough output to help, add a few debug output statements

```
cerr << "x == " << x << ", y == " << y << '\n’;
```

- Be very careful
  - See what the program specifies, not what you think it should say

```
That’s much harder to do than it sounds
for (int i=0; 0<month.size(); ++i) {  // oops!
for( int i = 0; i<=max; ++j) {        // oops! (twice)
```

Imperial College London

Debugging

- When you write the program, insert some checks (“sanity checks”) that variables have “reasonable values”
  - Function argument checks are prominent examples of this

```
if (number_of_elements<0)
     error("impossible: negative number of elements");

if (largest_reasonable<number_of_elements)
     error("unexpectedly large number of elements");

if (x<y) error("impossible: x<y");
```

- Design these checks so that some can be left in the program even after you believe it to be correct
  - It’s almost always better for a program to stop than to give wrong results

Imperial College London

Debugging

- Pay special attention to “end cases” (beginnings and ends)
  - Did you initialize every variable?
    - To a reasonable value
  - Did the function get the right arguments?
    - Did the function return the right value?
  - Did you handle the first element correctly?
    - The last element?
  - Did you handle the empty case correctly?
    - No elements
    - No input
  - Did you open your files correctly?
    - more on this in chapter 11
  - Did you actually read that input?
    - Write that output?

Imperial College London

Debugging

- "If you can't see the bug, you're looking in the wrong place"
  - It's easy to be convinced that you know what the problem is and stubbornly keep looking in the wrong place
  - Don't just guess, be guided by output
    - Work forward through the code from a place you know is right
      - so what happens next? Why?
    - Work backwards from some bad output
      - how could that possibly happen?
- Once you have found "the bug" carefully consider if fixing it solves the whole problem
  - It's common to introduce new bugs with a "quick fix"
- "I found the last bug"
  - is a programmer's joke

Imperial College London

Note

- Error handling is fundamentally more difficult and messy than "ordinary code"
  - There is basically just one way things can work right
  - There are many ways that things can go wrong
- The more people use a program, the better the error handling must be
  - If you break your own code, that's your own problem
    - And you'll learn the hard way
  - If your code is used by your friends, uncaught errors can cause you to lose friends
  - If your code is used by strangers, uncaught errors can cause serious grief
    - And they may not have a way of recovering

- Decide how to report errors: a good default answer to this question is: "Use `error()` and `catch exception&` in `main()`

Imperial College London

Pre-conditions

- What does a function require of its arguments?
  - Such a requirement is called a pre-condition
  - Sometimes, it's a good idea to check it

```
int area(int length, int width)  // calculate area of a rectangle
   // length and width must be positive
{
      if (length<=0 || width <=0) throw Bad_area{};
      return length*width;
}
```

Imperial College London

Post-conditions

- What must be true when a function returns?
  - Such a requirement is called a post-condition

```
int area(int length, int width)  // calculate area of a rectangle
   // length and width must be positive
{
      if (length<=0 || width <=0) throw Bad_area{};
      // the result must be a positive int that is the area
      // no variables had their values changed
      return length*width;
}
```

Imperial College
London

Pre- and post-conditions

- Always think about them
- If nothing else write them as comments
- Check them “where reasonable”
- Check a lot when you are looking for a bug
- This can be tricky
  - How could the post-condition for `area()` fail after the pre-condition succeeded (held)?

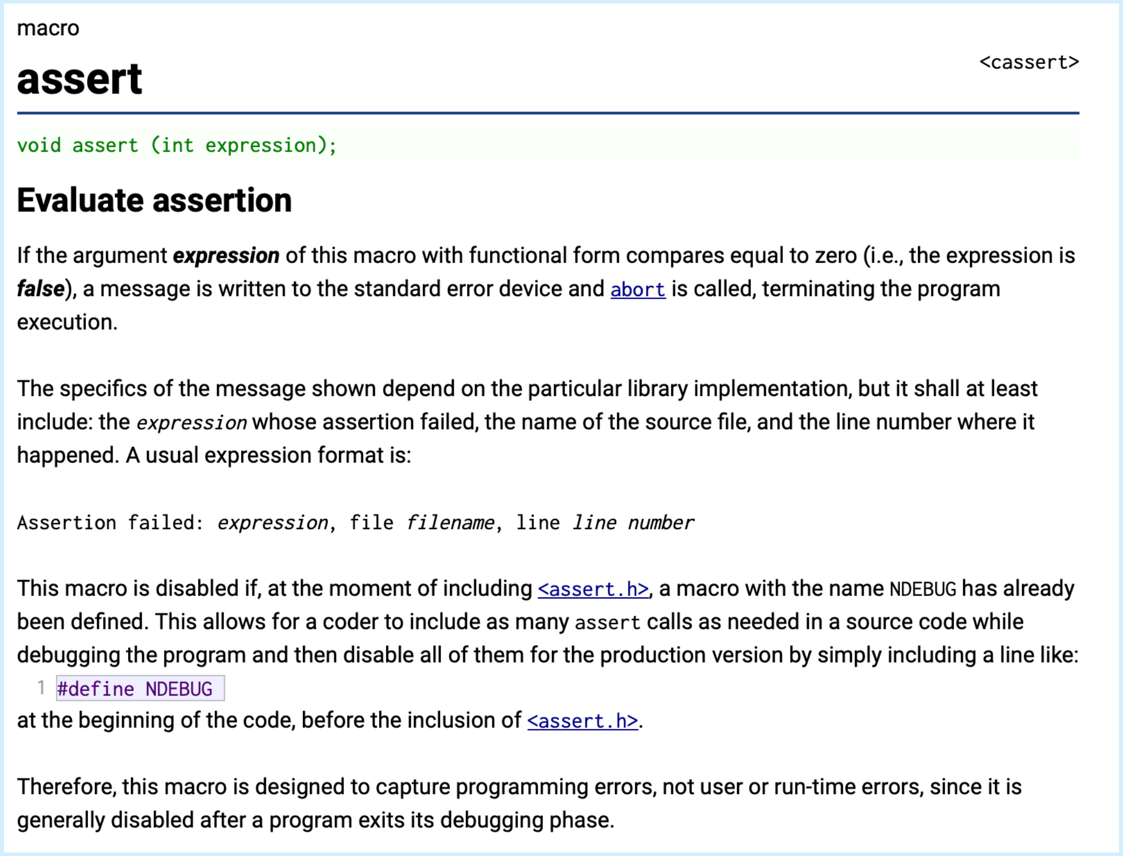
Testing

- How do we test a program?
  - Be systematic
    - “pecking at the keyboard” is okay for very small programs and for very initial tests, but is insufficient for real systems
  - Think of testing and correctness from the very start
    - When possible, test parts of a program in isolation
      - E.g., when you write a complicated function write a little program that simply calls it with a lot of arguments to see how it behaves in isolation before putting it into the real program (this is typically called “unit testing”)

Imperial College London

`assert()`

- Enlists the help of the debugger to monitor errors during runtime
- Only evaluated in DEBUG mode



macro

assert

<cassert>

```
void assert (int expression);
```

Evaluate assertion

If the argument ***expression*** of this macro with functional form compares equal to zero (i.e., the expression is ***false***), a message is written to the standard error device and abort is called, terminating the program execution.

The specifics of the message shown depend on the particular library implementation, but it shall at least include: the *expression* whose assertion failed, the name of the source file, and the line number where it happened. A usual expression format is:

```
Assertion failed: expression, file filename, line line number
```

This macro is disabled if, at the moment of including `<assert.h>`, a macro with the name `NDEBUG` has already been defined. This allows for a coder to include as many `assert` calls as needed in a source code while debugging the program and then disable all of them for the production version by simply including a line like:

```
#define NDEBUG
```

at the beginning of the code, before the inclusion of `<assert.h>`.

Therefore, this macro is designed to capture programming errors, not user or run-time errors, since it is generally disabled after a program exits its debugging phase.

Imperial College London

EXERCISE

- Write a function that divides one number by another one `a/b`,
  - `double divide(double a, double b)`
- Write a main that loops over 10,000 pairs of random doubles (between 0 and 1), divides them and accumulates the division into a variable `total_div`.
- Handle the possible error that can be caused by dividing by a zero value, using assert. Run your programme in Release and Debug mode. What is the difference?