Exceptions and RTTI; The Preprocessor

ITP 435
Week 3, Lecture 2

(Basic) Exception Sample

```
try
{
    unsigned int size = 1000000000;
    int* i = new int[size];
}
catch (std::bad_alloc&)
{
    std::cout << "Memory allocation failed :(" << std::endl;
}
catch (...) // Avoid catch (...) when possible
{
    std::cout << "Unknown exception??" << std::endl;
}
```

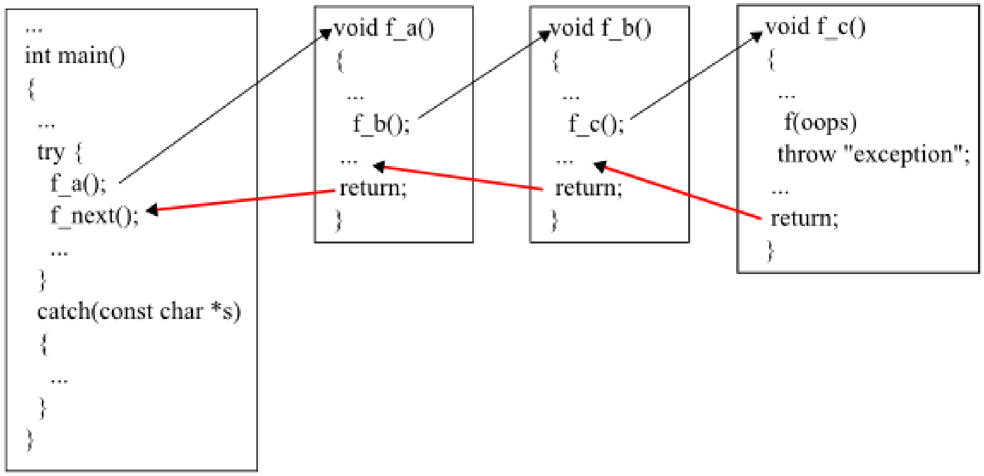
Why I *Usually* Don't Like Exceptions

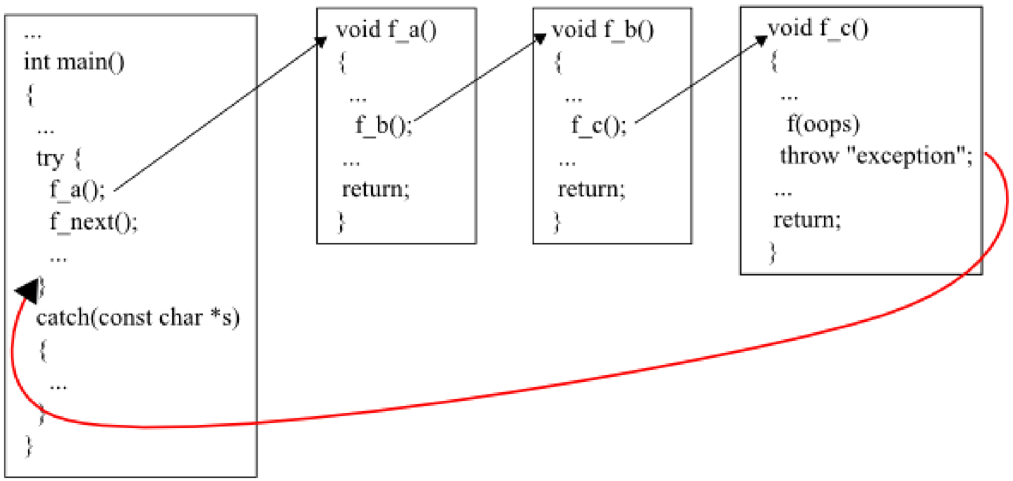
- This quote sums it up:

"In time-critical code, throwing an exception should *be* the exception, not the rule."

- Using exceptions have a memory cost and a runtime performance cost

Stack Unwinding





How’s this implemented?

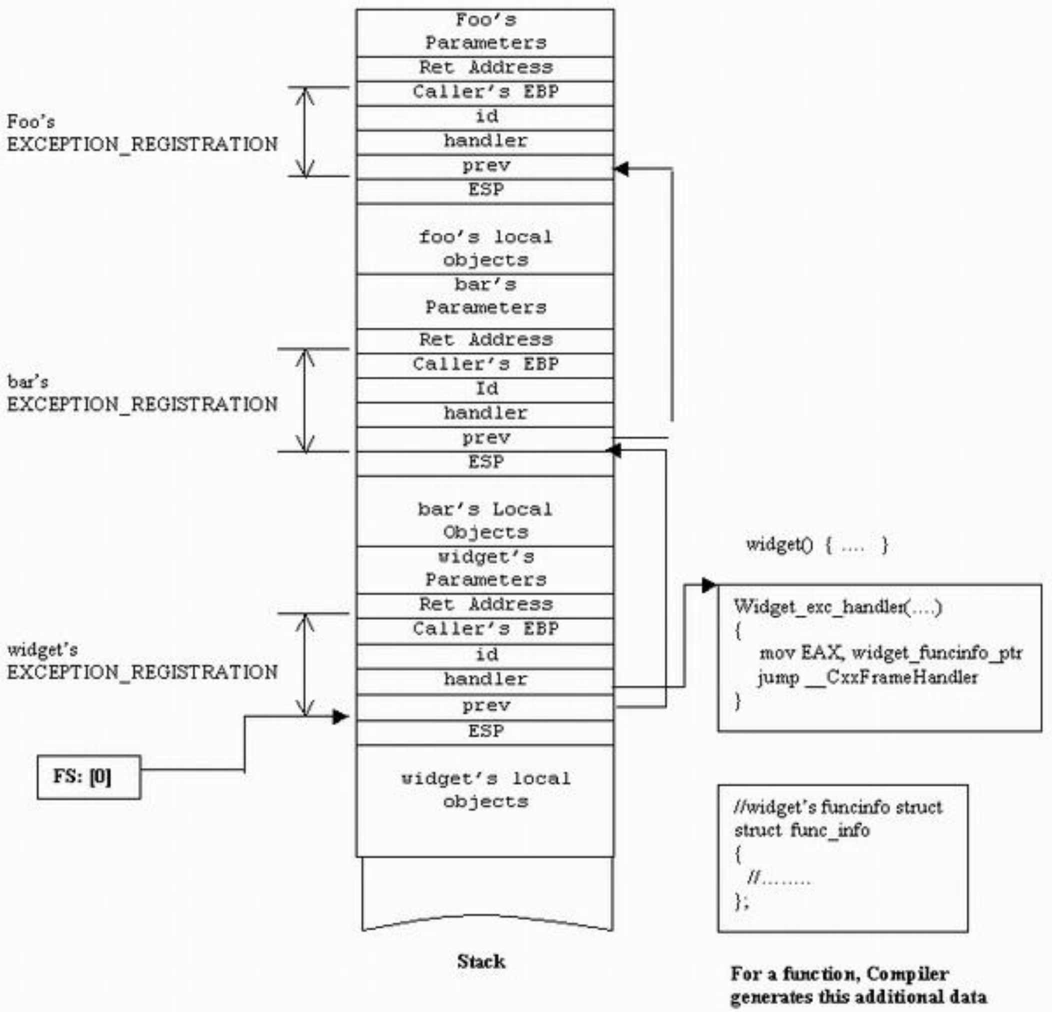


FIGURE 4

More Stack Unwinding

- If an exception is thrown, the destructor of any classes local to the try block will be called in reverse order.

```
try
{
	ClassA A();
	ClassB B();
	ClassC C();
	throw;
}
```

- When the throw happens, C will be destructed, then B, then A.

What class type to throw?

- Never, ever, ever throw a class type whose constructor could also throw an exception.
- Example:

```
try
{
    throw std::string("BAD EXCEPTION HERE");
}
```

- What happens if string's constructor throws an exception?
- Answer: The program terminates

std::exception

- Everything in the standard library throws an exception derived from `std::exception`…so you should also do this

```
#include <exception>
class MyException : public virtual std::exception
{
     // what is a virtual function which returns a description
     const char* what() const noexcept override { return "MY EXCEPTION!"; }
};
// later on...
try
{
     throw MyException();
}
catch (std::exception& e)
{
   std::cout << e.what() << std::endl; // "MY EXCEPTION!"
}
```

noexcept

- If you have a function that doesn't throw an exception, specify `noexcept`

```
// Default: Can throw anything
void Function1();

// Should not throw an exception
void Function2() noexcept;
```

- NOTE: You have to specify `noexcept` on an `override` of "what"

Unexpected Exceptions!

- What happens when you throw an exception from a `noexcept` function?

- `std::terminate`.

catch (...)

- Problem with `catch (...)` is you could get crazy exceptions from the operating system ☹

- Added bonus of deriving from `std::exception`…
- You can avoid `catch (...)`
- Instead we can use `catch (std::exception& e)`

Exception-Safe Code

- Any code within a `try` block can't assume it succeeds.
- Therefore, you can't leave any dangling resources.
- Example from Effective C++:

```
void PrettyMenu::changeBackground(std::istream& imgSrc)
{
    lock(&mutex);
    delete bgImage;
    ++imageChanges;
    bgImage = new Image(imgSrc);
    unlock(&mutex);
}
```

Solution: use ***RAII***

- What happens if `new` throws an exception?

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Let's assume the entry point is fa

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Calls fb...

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Calls fc…

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

fc throws an exception.

Is it caught here?

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

fc throws an exception.

Is it caught here?

No! – So unwind the call stack

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

We're now back in fb's scope.

Before checking the catches, ***destruct*** `myAA2`!

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

No! – So unwind the call stack

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

We're now back in fa's scope.

Before checking the catches, ***destruct*** `myAA`!

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

A More In-depth Stack Unwinding Example

```
class AA {};
struct ExceptThis : public std::exception {};
void fa() {
   try {
      AA myAA;
      fb();
   }
   catch (std::exception& e) {
      std::cout << "Uh oh\n";
   }
}
void fb() {
   try {
      AA myAA2;
      fc();
   }
   catch (const char* e) {
      std::cout << "Not good\n";
   }
}
void fc() {
   throw ExceptThis();
}
```

Is it caught here?

Yes! – because `ExceptThis` inherits from `std::exception`

In-class Activity

- Let’s look more at exceptions

Exception Summary

- If you're going to use exceptions, follow these rules:

1. Always throw an exception derived from `std::exception`
2. Never use `catch(...)`
3. Use `noexcept` when appropriate
4. Refactor code so it's exception-safe, as necessary

RTTI

- RTTI = ***Run-time Type Information*** (or Run-time Type Identification)
- In order for exceptions to work, C++ needs to figure out, at run-time, the type of the exception
- Thus, RTTI and exceptions sort of go hand in hand.

- RTTI ***ONLY*** works properly for classes that are polymorphic (eg. They have at least one virtual function, inherited or not).
- This is because RTTI information is stored in the virtual function table

dynamic_cast

- A down cast allows you to take a parent class pointer, and at runtime try to cast it to a child class.

- Eg. If you have a "Shape" pointer, and want to find out if it's a "Triangle" at runtime, you can do:

```
Shape* myShape;
Triangle* myTriangle = dynamic_cast<Triangle*> (myShape);
if (myTriangle) // dynamic_cast returns 0 if not a triangle
{
     // do something
}
```

- Should only be used in cases where you have a function you don't want to expose in the base class, which is rare.

typeid

- Allows you to figure out the type of something at runtime

```
class Person
{
public:
	virtual ~Person() {}
};
class Employee : public Person
{
};

// Later...
Person* ptr = new Employee();
if (typeid(*ptr) == typeid(Employee))
{
	// This is an employee!!
}
```

std::type_info

- The use of `typeid` returns a `std::type_info` class, which is defined in `<typeinfo>`

- Notable member functions:
  - `operator!=` (Self-explanatory)
  - `operator==` (Self-explanatory)
  - name (Returns implementation-specific name, as a const char*)

typeid().name()

- Returns a char* with the name of the type

```
// Outputs "class Employee"
// Note output string is compiler-dependant
std::cout << typeid(*ptr).name() << std::endl;
```

typeid gotcha

- If you use typeid on a *non-polymorphic* (eg. no virtual function) class, it wont' work as you might expect it to:

```
class Person
{
public:
//      virtual ~Person() {}
};
class Employee : public Person
{
};

// Later...
Person* ptr = new Employee();

// Outputs "class Person"
std::cout << typeid(*ptr).name() << std::endl;
```

Default RTTI Drawback

- Every single class with a virtual function has additional RTTI information stored in its virtual function table
- What happens if you have 10,000 classes with virtual functions, but you only need RTTI for 100 of them?
- Answer: Unnecessary memory waste
- Some C++ libraries (LLVM, for example) implement their own RTTI for this reason

LLVM RTTI – Why?

In an effort to reduce code and executable size, LLVM does not use RTTI (e.g. `dynamic_cast<>;`) or exceptions. These two language features violate the general C++ principle of “you only pay for what you use”, causing executable bloat even if exceptions are never used in the code base, or if RTTI is never used for a class. Because of this, we turn them off globally in the code.

That said, LLVM does make extensive use of a hand-rolled form of RTTI that use templates like `isa<>`, `cast<>`, and `dyn_cast<>`. This form of RTTI is opt-in and can be added to any class. It is also substantially more efficient than `dynamic_cast<>`.

Our Own Custom RTTI

- Step 1: Declare a “type info” class…

```
class TypeInfo {
public:
    // Takes pointer to super class' type info
    TypeInfo(const TypeInfo* super) : mSuper(super) {}
    // Is this type exactly matching the other pointer?
    bool IsExactly(const TypeInfo* other) const {
        return (this == other);
    }
    // Return the super type
    const TypeInfo* SuperType() const {
        return mSuper;
    }
private:
    const TypeInfo* mSuper;
};
```

Our Own Custom RTTI, cont'd

- Step 2: For hierarchies that use it, declare the following:

```
class Object {
private:
   static const TypeInfo sType;
public:
   static const TypeInfo* StaticType() { return &sType; }
   virtual const TypeInfo* GetType() const { return &sType; }
};
```

- Initialize as follows (if base class):

```
const TypeInfo Object::sType(nullptr);
```

- A derived class would look like this:

```
const TypeInfo Derived::sType(SuperClass::StaticType());
```

Idea: Declare macros to make life easier…

```
#define DECL_OBJECT()  \
    private: \
    static const TypeInfo sType; \
    public: \
    static const TypeInfo* StaticType() { return &sType; } \
    const TypeInfo* GetType() const override { return &sType; } \


#define IMPL_OBJECT(d,s) \
    const TypeInfo d::sType(s::StaticType()); \
```

…then can use it like this

```
class Derived : public Object
{
    DECL_OBJECT();
};

IMPL_OBJECT(Derived, Object);
```

Is-A Function

```
// Returns true if ptr is-a Type
// Usage: IsA<Type>(ptr)
template <typename Other, typename This>
bool IsA(const This* ptr) {
   for (const TypeInfo* t = ptr->GetType();
           t != nullptr;
           t = t->SuperType()) {
      if (t->IsExactly(Other::StaticType())) {
         return true;
      }
   }
   return false;
}
```

Cast Function

```
// Casts ptr to Type if valid
// Usage: Cast<Type>(ptr)
template <typename Other, typename This>
Other* Cast(This* ptr) {
    if (IsA<Other>(ptr)) {
        return static_cast<Other*>(ptr);
    }
    else {
        return nullptr;
    }
}
```

Note

- Our RTTI implementation assumes single-inheritance…multiple inheritance gets messier

- This covers the approach used by LLVM (which does work for multiple inheritance):

http://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html

The Preprocessor

Preprocessor

- Processes all # directives to generate the final C++ code which will be compiled
- The resulting code is often called a "translation unit"
- Example 1:

```
#include "dbg_assert.h"
// dbg_assert.h code is essentially copy/pasted at this line
```

- Example 2:

```
// Compile this only in a "debug" build
#ifdef _DEBUG
// Random debug code…
#endif
```

- Example 3:

```
// Replaces "MAX_POOL_SIZE" with 256 in code
// (Breaks Rule #2 in Effective C++)
#define MAX_POOL_SIZE 256
```

Be careful with #include

- Don't make an "everything.h" that you include everywhere:

```
// INCLUDE EVERYTHING
#include <algorithm>
#include <bitset>
#include <cassert>
#include <cctype>
#include <cerrno>
#include <cfloat>
// ...
```

- Only include files you really need to include!

Include “Guard”

- May have seen this before:

```
#ifndef _MYFILE_H_
#define _MYFILE_H_

// stuff here

#endif // _MYFILE_H_
```

- The above works, but my preferred method is to put this at the start of the header (works in Visual Studio, Clang, and GCC):

```
#pragma once
```

Macros

- Not only can we define values like this:

```
#define MY_VALUE 10
```

- We can replace one expression with another:

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

- So if you write code like this:

```
std::cout << max(5, 6);
```

- Preprocessor replaces max with the defined code and our parameters:

```
std::cout << (((5) > (6)) ? (5) : (6));
```

Macros, Part 2

- Problem 1: Macros can clash with other functions/classes with confusing errors.
- What if I later declare…

```
void max();
```

- Error messages:

```
warning C4003: not enough actual parameters for macro 'max'
error C2059: syntax error : ')'
error C2059: syntax error : ')'
error C2059: syntax error : ')'
```

Macros, Part 3

- Problem 2: Must be very careful with parenthesis

```
#define MULT(x, y) x * y
// What if I do this?
int z = MULT(3 + 2, 4 + 2);
```

- Preprocessor evaluates it to:

```
int z = 3 + 2 * 4 + 2;
```

- Instead, you need a lot of parenthesis

```
#define MULT(x, y) ((x) * (y))
```

- So the preprocessor gives you:

```
int z = ((3 + 2) * (4 + 2));
```

Macro Counter-Point

- Instead of this (which is type-agnostic)

```
#define max(a,b) (((a) > (b)) ? (a) : (b))
```

- C++ lets us do (which is also type-agnostic):

```
template <class T>
T max(T a, T b)
{
    return ((a > b) ? a : b);
}
```

Macro Counter-Point Counter-Point

- What about something like this:

```
class CBlackjackView : public CWindowImpl<CBlackjackView>
{
public:
   DECLARE_WND_CLASS(NULL)

   BEGIN_MSG_MAP(CBlackjackView)
    MESSAGE_HANDLER(WM_PAINT, OnPaint)
    MESSAGE_HANDLER(WM_CREATE, OnCreate)
   END_MSG_MAP()
// ...
};
```

DECLARE_WND_CLASS Macro

```
#define DECLARE_WND_CLASS(WndClassName) \
static ATL::CWndClassInfo& GetWndClassInfo() \
{ \
static ATL::CWndClassInfo wc = \
{ \
{ sizeof(WNDCLASSEX), CS_HREDRAW | CS_VREDRAW | CS_DBLCLKS,
   StartWindowProc, \
  0, 0, NULL, NULL, NULL, (HBRUSH)(COLOR_WINDOW + 1), NULL,
   WndClassName, NULL }, \
NULL, NULL, IDC_ARROW, TRUE, 0, _T("") \
}; \
return wc; \
}
```

Preprocessor Trick – Stringify

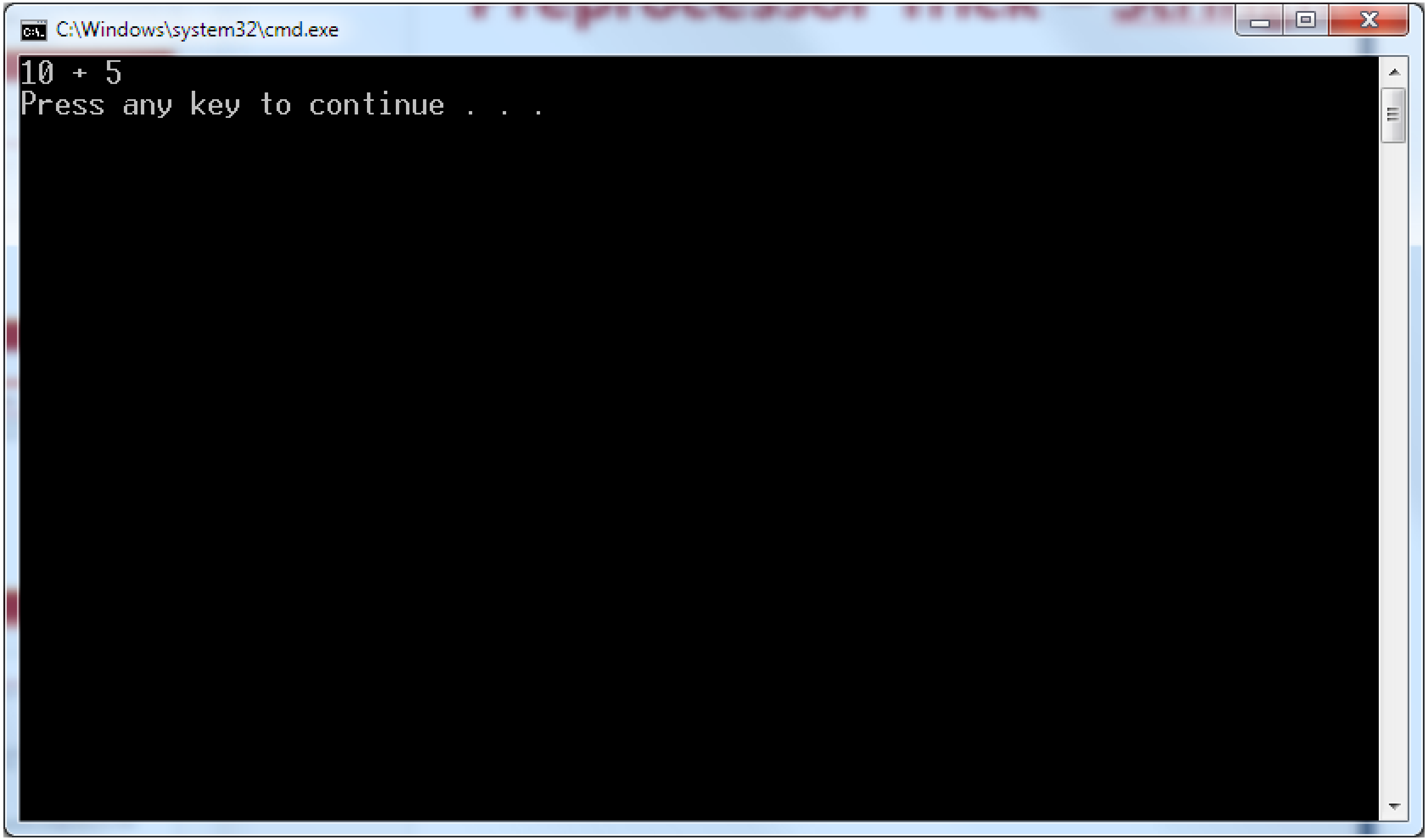
- You can convert any token passed to the preprocessor to a string using # in front of the parameter name

```
#include <iostream>

#define TO_STRING(str) #str

int main() {
    std::cout << TO_STRING(10 + 5) << std::endl;
    return 0;
}
```

Stringify in Action



Token Concatenation

- You can use ## to concatenate a preprocessor token to a set value, for example:

```
#include <iostream>

#define DECLARE_VAR(var) static int var##_s = 5;

int main() {
   DECLARE_VAR(hello);
   std::cout << hello_s << std::endl;
   return 0;
}
```

X-Macros

- One very useful (but advanced) macro design pattern is X-Macros

- An ***X-Macro*** can be used to generate a list of repetitive code constructs at preprocessor time

- Can save a lot of annoying repetition, though they are a little confusing to use

An example – Tokens.def

```
// Expression Operators
TOKEN(Assign,"=",1)
TOKEN(Plus,"+",1)
TOKEN(Minus,"-",1)
TOKEN(Mult,"*",1)
TOKEN(Div,"/",1)
TOKEN(Mod,"%",1)
TOKEN(Inc,"++",2)
TOKEN(Dec,"--",2)
TOKEN(LBracket,"[",1)
TOKEN(RBracket,"]",1)
TOKEN(EqualTo,"==",2)
TOKEN(NotEqual,"!=",2)
TOKEN(Or,"||",2)
TOKEN(And,"&&",2)
TOKEN(Not,"!",1)
TOKEN(LessThan,"<",1)
TOKEN(GreaterThan,">",1)
TOKEN(LParen,"(",1)
TOKEN(RParen,")",1)
TOKEN(Addr,"&",1)
```

Using Tokens.def to Generate an enum…

```
enum Tokens
{
    #define TOKEN(a,b,c) a,
    #include "Tokens.def"
    #undef TOKEN
};
```

Using Tokens.def to generate arrays of data

```
static const char* Names_data[] =
{
    #define TOKEN(a,b,c) #a,
    #include "Tokens.def"
    #undef TOKEN
};

static const char* Values_data[] =
{
    #define TOKEN(a,b,c) b,
    #include "Tokens.def"
    #undef TOKEN
};

static const int Lengths_data[] =
{
    #define TOKEN(a,b,c) c,
    #include "Tokens.def"
    #undef TOKEN
};
```