Chapter 8: Technicalities: Functions, etc. (continues)

Plan for today

- We will talk about:
  - Function call implementation
  - Compile time functions
  - Namespaces

Function call implementation

- How does a computer do a function call?

Function call implementation

- How does a computer do a function call?
- When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and local variables

Function call implementation

- How does a computer do a function call?
- When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and local variables
- Such a data structure is called a *function activation record*

Function call implementation

- How does a computer do a function call?
- When a function is called, the language implementation sets aside a data structure containing a copy of all its parameters and local variables
- Such a data structure is called a *function activation record*
- Each function has its own detailed layout of its activation record

Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:

call stack

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

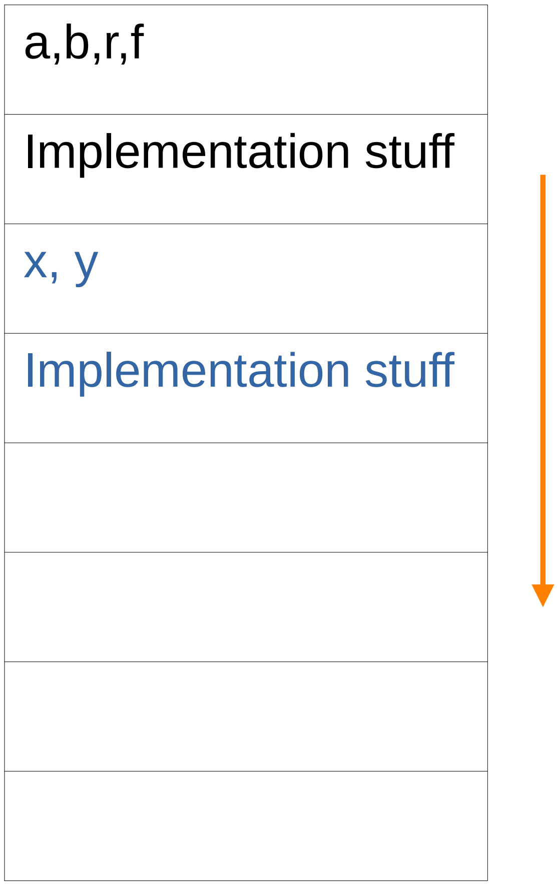
Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of `main()`:

call of `f2()`:

call stack



a,b,r,f

Implementation stuff

x, y

Implementation stuff

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

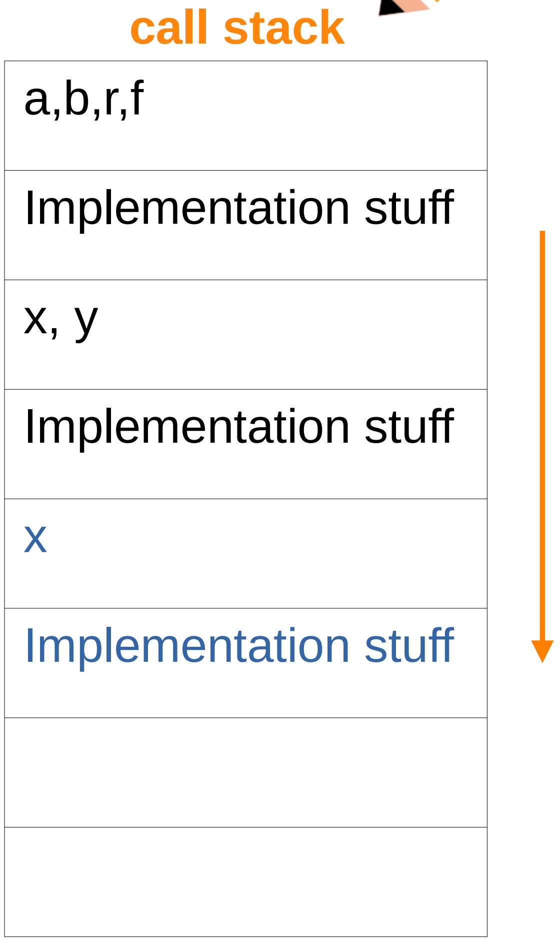
Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:

call of **`f2()`**:

call of **`f1()`**:



call stack

a,b,r,f

Implementation stuff

x, y

Implementation stuff

x

Implementation stuff

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

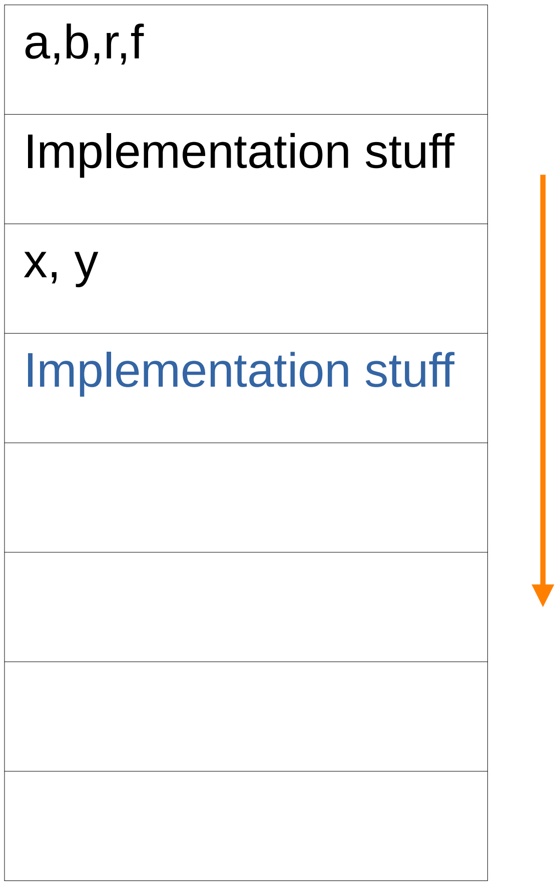
Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:

call of **`f2()`**:

call stack



call stack
a,b,r,f
Implementation stuff
x, y
Implementation stuff

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

Function call implementation: example

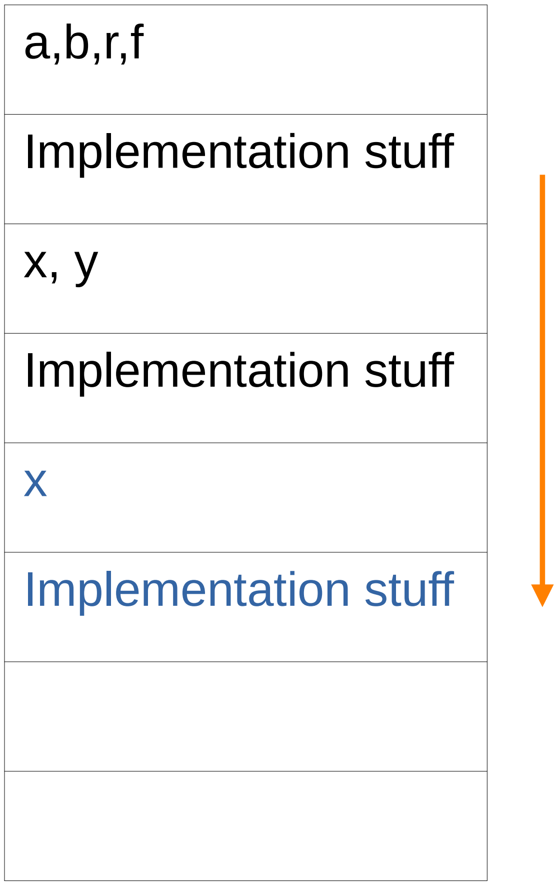
```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of `main()`:

call of `f2()`:

call of `f3()`:

call stack



a,b,r,f

Implementation stuff

x, y

Implementation stuff

x

Implementation stuff

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

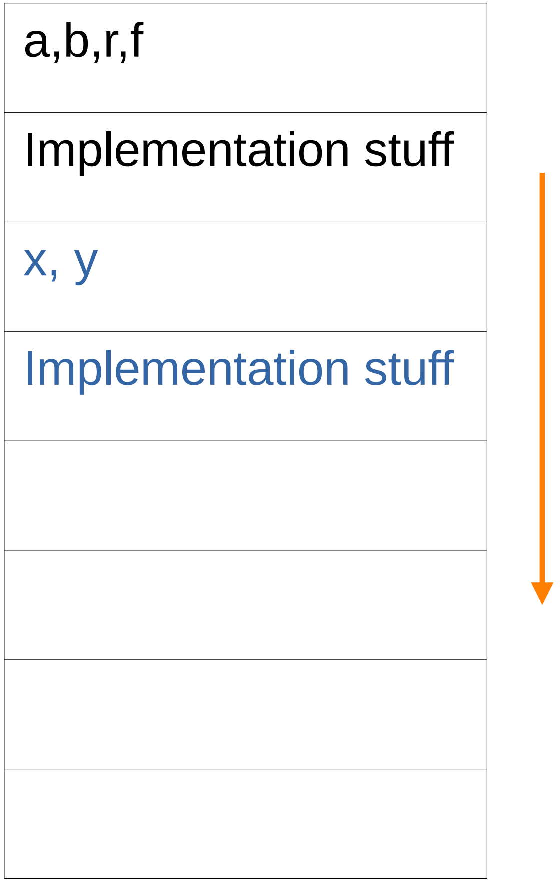
Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:

call of **`f2()`**:

call stack



Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:

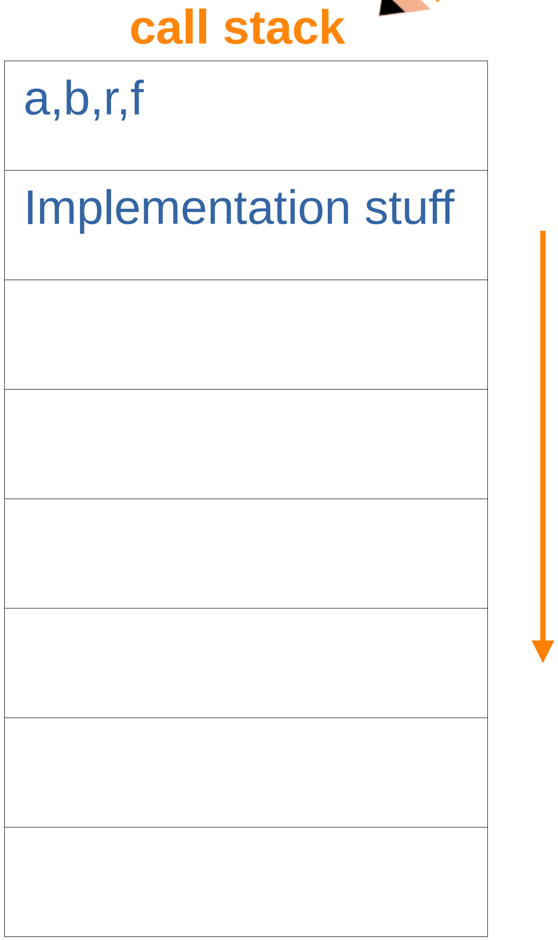
call stack

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

call of **`main()`**:



Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

Function call implementation: example

```
int main() {
   int a{ 3 }, b{ -5 }, r;
   r = f2(a, b);
   cout << "result r = " << r;
}

int f1(int x) {
   return x * x;
}
int f2(int x, int y) {
   return f1(x) + f3(y);
}
int f3(int x) {
   return x * x * x;
}
```

Implementation stuff *has information that the function needs to return to its caller and to return a value to its caller.*

`constexpr` functions

- Sometimes we want to do a calculation at compile time
  - usually to avoid having the same calculation done many many times at run time
- By declaring the function `constexpr`, and providing constant expressions as argument, we convey our intent to the compiler
- In addition, a `constexpr` function may not have side effects, i.e. may not change the value of variables outside its own body
- Should return a value (starting from C++ 11)
- May have a simple loop (starting from C++ 14)
- Compiler can evaluate such a function at compile time

constexpr functions

```
constexpr double xscale = 10;    // scaling factor
constexpr double yscale = .8;   // scaling factor

constexpr Point scale(Point p) {
   return { xscale*p.x, yscale*p.y }; }

constexpr Point x = scale({123,456}); // evaluated at compile time

void use(Point p)
{
   constexpr Point x1 = scale(p);  // error: compile-time
                 // evaluation requested for variable argument
   Point x2 = scale(p);     // OK: run-time evaluation
}
```

refer to program **CompileTimeExamples.cpp** for more details

constexpr functions

```
double x = 10;    // global variable

constexpr void func(int &arg) // no return value
{
   ++arg;   // error: modifies caller via argument
    x = 2.7; // error: modifies nonlocal variable
}
```

- this is an example of a function that violates rules for simplicity.

Global initialization

- Global variables in a single translation unit are initialized in the order in which they appear
- Using a global variable is usually not a good idea
  - no really effective way of knowing which parts of a large program reads/writes global variable
  - the order of initialization of global variables in different translation units is not defined

Global initialization

- Global variables <u>in a single translation unit</u> are initialized in the order in which they appear
- Using a global variable is usually not a good idea
  - no really effective way of knowing which parts of a large program reads/writes global variable
  - the order of initialization of global variables in different translation units is not defined

In file f1.cpp:

```
int x1 = 1;
int y1 = x1 + 2;
```

in file f2.cpp:

```
extern y1;
int y2 = y1 + 2;
```

Global initialization

- Global variables <u>in a single translation unit</u> are initialized in the order in which they appear
- Using a global variable is usually not a good idea
  - no really effective way of knowing which parts of a large program reads/writes global variable
  - the order of initialization of global variables in different translation units is not defined

In file f1.cpp:

```
int x1 = 1;
int y1 = x1 + 2;
```

in file f2.cpp:

```
extern y1;
int y2 = y1 + 2;
```

avoid using short names and complicated initialization.

Global initialization: real need

- What to do if we really need a global variable (or a constant) with a complicated initializer?
- An example: a default value for a **Date** type

```
const Date default_date(1970,1,1); // Jan. 1st, 1970
```

How would we know that `default_date` was never used before it was initialized?

Basically, we can’t know, so we shouldn’t write that definition.

Global initialization: real need

- The technique that we use most oftern is to call a function that returns a value
- An example: a default value for a **Date** type

```
const Date default_date() // return Jan. 1st, 1970
{
   return Date(1970,1,1);
}
```

This constructs the Date every time we call `default_date()`.

If it is called frequently, then it becomes expensive. In this case, if we want the construct the default **Date** only once, we can use **static** variable.

Global initialization: real need

- Using static variable to have only one **Date** object for default value:

```
const Date& default_date() // return Jan. 1st, 1970
{
   static const Date dd(1970,1,1); // initialize
                                    // dd first time we get here
   return dd;
}
```

Namespaces

- We use blocks to organize code within a function
- We use classes to organize functions, data and types into a type
- A function and a class both do two things for us:
  - they allow us to define a number of ”entities” without worrying that their names clash with other names in our program
  - they give us a name to refer to what we have defined
- We need something to organize classes, functions, data and types into an identifiable and named part of a program without defining a type
  - The language mechanism for such grouping is a *namespace*.

Namespaces

Consider this code from two programmers Jack and Jill:

```
   class Glob { /*…*/ };          // in Jack’s header file jack.h
   class Widget { /*…*/ };       // also in jack.h

   class Blob { /*…*/ };          // in Jill’s header file  jill.h
   class Widget { /*…*/ };       // also in jill.h
```

```
   #include "jack.h";  // this is in your code
   #include "jill.h";  // so is this

   void my_func(Widget p) //oops! error: multiple definitions of
Widget
   {
      // …
   }
```

Namespaces

- The compiler will not compile multiple definitions; such clashes can occur from multiple headers.
- One way to prevent this problem is with namespaces:

```
namespace Jack {    //  in Jack’s header file
     class Glob{ /*…*/ };
      class Widget{ /*…*/ };
   }

 #include "jack.h";  // this is in your code
#include "jill.h";   // so is this

void my_func(Jack::Widget p)  // OK, Jack’s Widget class will not
{    // clash with a different Widget
     // …
}
```

Namespaces

- A *namespace* is a named scope
- The **::** syntax is used to specify which namespace you are using and which (of many possible) objects of the same name you are referring to – often called “scope resolution”
- For example, **cout** is in **namespace std**, you could write:

```
std::cout << "Please enter stuff… \n";
```

using Declarations and Directives

- To avoid the tedium of
  - `std::cout << "Please enter stuff… \n";`

  you could write a “using declaration”
  - `using std::cout;  // when I say cout, I mean std::cout`
  - `cout << "Please enter stuff… \n";  // ok: std::cout`
  - `cin >> x; // error: cin not in scope`
- or you could write a “using directive”
  - `using namespace std;` // “make all names from namespace std available”
  - `cout << "Please enter stuff… \n";` // ok: `std::cout`
  - `cin >> x;` // ok: `std::cin`
- More about header files in chapter 12

In-class work

- Let’s get back to that in-class work from the previous meeting:

  grab the suggested code for the statistics program and incorporate a few suggestions into it:

  - We want to keep sorting, finding minimum, maximum, mean and median as one function, but we want to be able to return the smallest, the largest, the mean and the median back to the caller. What can we do?
  - We would also like to make “input” as a separate function.

Resources used for these slides

- slides provided by B. Stroustrup at https://www.stroustrup.com/PPP2slides.html
- Class textbook