Imperial College London

Advanced Programming

C++ Basic Structure, Declarations & Definitions

Adriana Paluszny

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

Imperial College London

Brief Quiz

- Go to **www.menti.com** and use the code **1366 6611**

Mentimeter

Imperial College London

Computation

- We aim to express computations
  - Correctly
  - Simply
  - Efficiently
- One strategy is “Divide and Conquer”
  - to break up big computations into many little ones
- Another strategy is “Abstraction”
  - Provide a higher-level concept that hides detail
- Organisation of data is often the key to good code
  - Input/output formats
  - Protocols
  - Data structures
- Emphasis on structure and organisation
  - You don’t get good code just by writing a lot of statements

Imperial College London

Expressions

```
// compute area:
int length = 20;          // the simplest expression: a literal (here, 20)
                          // (here used to initialize a variable)
int width = 40;
int area = length*width;           // a multiplication
int average = (length+width)/2;  // addition and division
```

The usual rules of precedence apply:

`a*b+c/d` means **`(a*b)+(c/d)`** and not **`a*(b+c)/d.`**

If in doubt, parenthesize. If complicated, parenthesize.
Don't write "absurdly complicated" expressions:

```
a*b+c/d*(e-f/g)/h+7        // too complicated
```

Choose meaningful names.

Expressions

Expressions are made out of operators and operands

- Operators specify what is to be done
- Operands specify the data for the operators to work with

Boolean type: **bool** (**true** and **false**)

- Equality operators: **= =** (equal), **!=** (not equal)
- Logical operators: **&&** (and), **||** (or), **!** (not)
- Relational operators: **<** (less than), **>** (greater than), **<=**, **>=**

Character type: **char** (e.g., **'a'**, **'4'**, and **'@'**)

Integer types: **short, int, long**

- arithmetic operators: **+, -, *, /, %** (remainder)

Floating-point types: e.g., **float, double** (e.g., **12.45** and **1.234e3**)

- arithmetic operators: **+, -, *, /**

Concise Operators

For many binary operators, there are (roughly) equivalent concise operators

– For example

`a += c`	means	`a = a+c`
`a *= scale`	means	`a = a*scale`
`++a`	means	`a += 1`
	or	`a = a+1`

– “Concise operators” are generally better to use (clearer, express an idea more directly)

Imperial College London

Statements

- A statement is
  - an expression followed by a semicolon, or
  - a declaration, or
  - a “control statement” that determines the flow of control

- For example

```
a = b;
double d2 = 2.5;
if (x == 2) y = 4;
while (cin >> number) numbers.push_back(number);
int average = (length+width)/2;
return x;
```

- You may not understand all of these just now, but you will …

Imperial College London

Selection

- Sometimes we must select between alternatives
- For example, suppose we want to identify the larger of two values. We can do this with an **if** statement

```
if (a<b)     // Note:  No semicolon here
   max = b;
else         // Note:  No semicolon here
   max = a;
```

- The syntax is

```
if (condition)
    statement-1  // if the condition is true, do statement-1
else
    statement-2  // if not, do statement-2
```

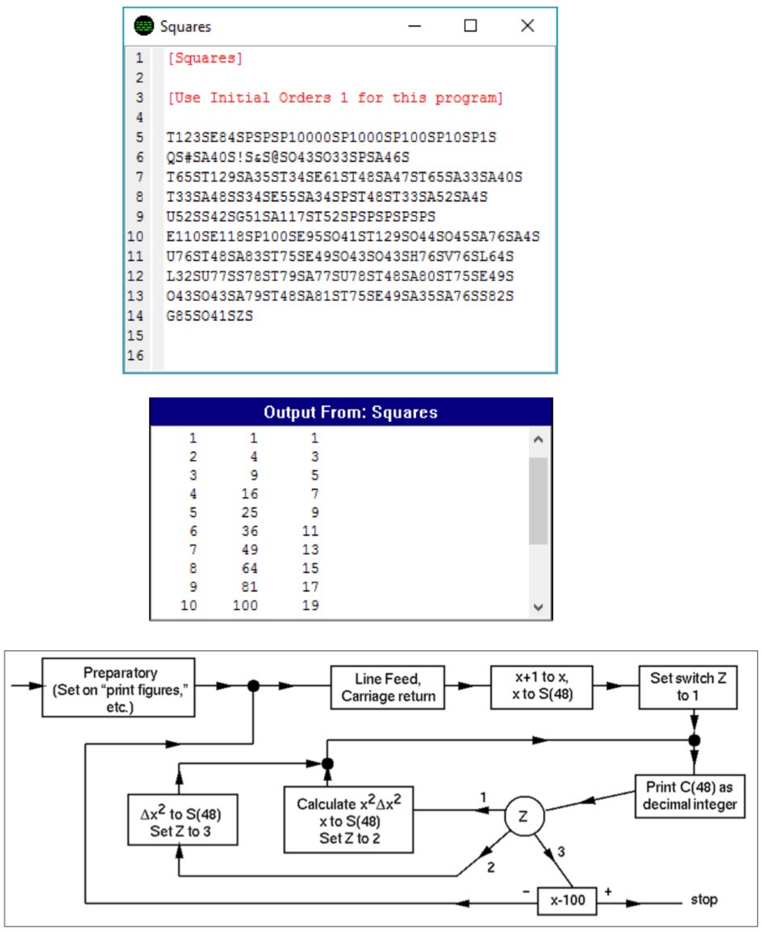
Imperial College London

Iteration (while loop)

- The world’s first “real program” running on a stored-program computer (David Wheeler, Cambridge, May 6, 1949)

```
// calculate and print a table of squares 0-99:
int main()
{
    int i = 0;
    while (i<100) {
      cout << i << '\t' << square(i) << '\n';
      ++i ; // increment i
    }
}
```

(the original was written in assembly)`



(a) Program manuscript, *right* (b) Program tape, *above top*

(c) Printout, *above middle* (d) Flow-diagram, *above*

Iteration (while loop)

What it takes

- A loop variable (control variable); here: `i`
- Initialize the control variable; here: `int i = 0`
- A termination criterion; here: `if i<100` is false, terminate
- Increment the control variable; here: `++i`
- Something to do for each iteration; here: `cout << …`

```
int i = 0;
while (i<100)  {
  cout << i << '\t' << square(i) << '\n';
  ++i ;   // increment i
}
```

Iteration (for loop)

Another iteration form: the `for` loop

You can collect all the control information in one place, at the top, where it's easy to see

```
for (int i = 0; i<100; ++i) {
  cout << i << '\t' << square(i) << '\n';
}
```

That is,

```
for (initialize; condition ; increment )
controlled statement
```

Note: what is **square(i)**?


TIME FOR A BREAK!

Imperial College London

Functions

But what was `square(i)`?

- A call of the function `square()`

```
int square(int x)
{
   return x*x;
 }
```

- We define a function when we want to separate a computation because it
  - is logically separate
  - makes the program text clearer (by naming the computation)
  - is useful in more than one place in our program
  - eases testing, distribution of labor, and maintenance

Imperial College London

Control Flow

```
int main()
{
    i=0;
    while (i<100)
    {
        square(i);
    }
}
```

```
int square(int x)
{
    // compute square
    return x*x;
}
```

Functions

Our function

```
int square(int x)
{
  return x*x;
}
```

is an example of

```
Return_type  function_name ( Parameter list )
                          // (type name, etc.)
{
  // use each parameter in code
  return some_value;     // of Return_type
}
```

Another Example

Earlier we looked at code to find the larger of two values. Here is a function that compares the two values and returns the larger value.

```
int max(int a, int b)  // this function takes 2 parameters
{
    if (a<b)
       return b;
    else
       return a;
}

int x = max(7, 9);         // x becomes 9
int y = max(19, -27);      // y becomes 19
int z = max(20, 20);       // z becomes 20
```

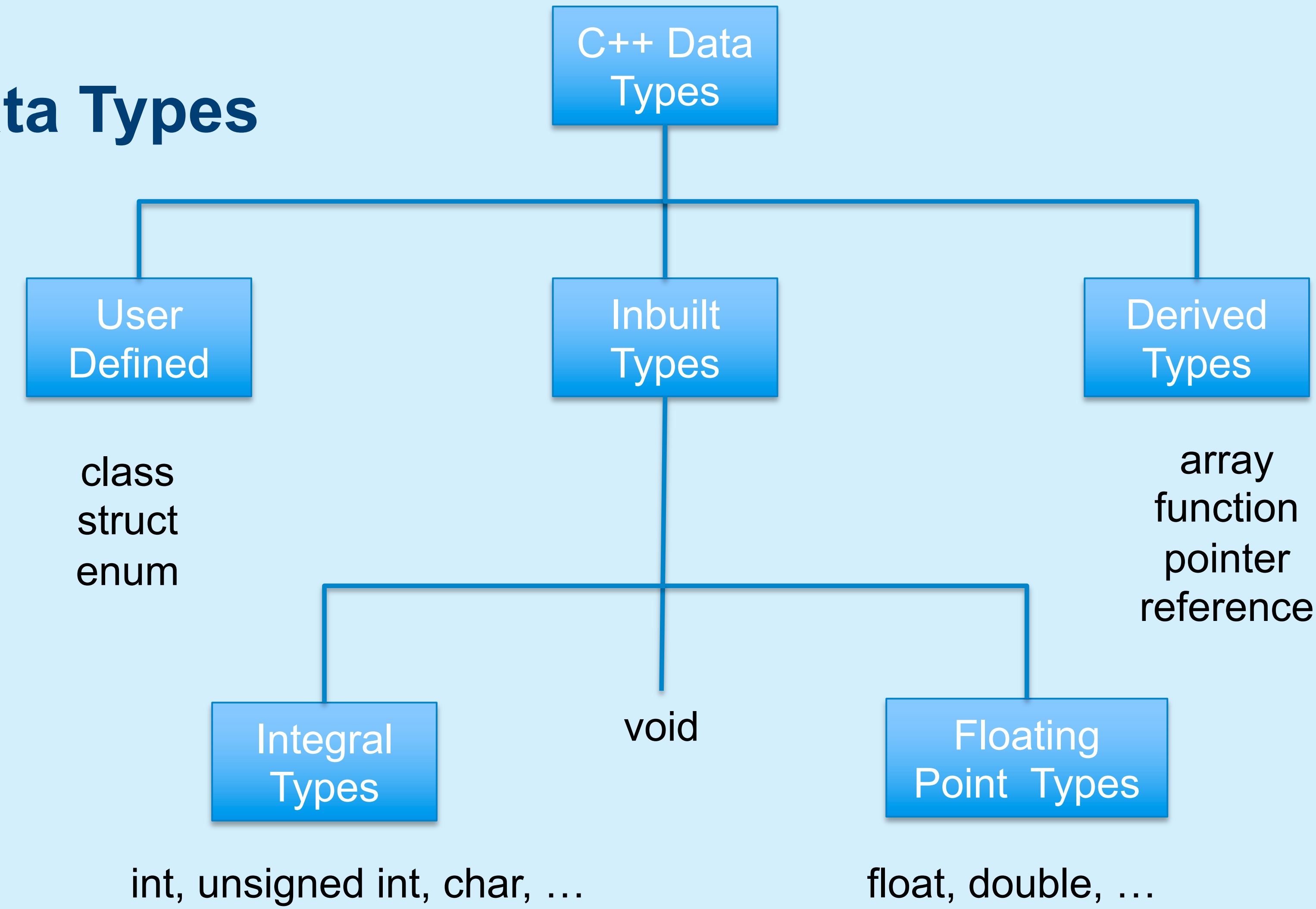
Language technicalities

- Technical terms are necessary
  - A programming language is a foreign language
  - When learning a foreign language, you have to look at the grammar and vocabulary
  - We will do this indirectly
- Because:
  - Programs must be precisely and completely specified
  - It is important to understand the rules
    - Some of them (the C++23 standard working draft is 2,110 pages)
- However, never forget that
  - What we study is programming
  - Our output is programs/systems
  - A programming language is only a tool

https://isocpp.org/files/papers/N4928.pdf

Imperial College London

Data Types



Imperial College London

In-built data types have bounds

Type Name	Kind of Value	Memory Used	Range of Values
byte	Integer	1 byte	−128 to 127
short	Integer	2 bytes	−32,768 to 32,767
int	Integer	4 bytes	−2,147,483,648 to 2,147,483,647
long	Integer	8 bytes	−9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
float	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
double	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
char	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
boolean		1 bit	True or false

Some thoughts

- Don’t spend your time on minor syntax and semantic issues. There is more than one way to say everything
  - Just like in English
- Most design and programming concepts are universal, or at least very widely supported by popular programming languages
  - So what you learn using C++ you can use with many other languages
- Language technicalities are specific to a given language
  - But many of the technicalities from C++ presented here have obvious counterparts in C, Java, C#, etc.

Declarations

- A declaration introduces a name into a scope.
- A declaration also specifies a type for the named object.
- Sometimes a declaration includes an initializer.
- A name must be declared before it can be used in a C++ program.
- Examples:

```
int z = 7;                     // an int variable named ‘z’ is declared
const double cd = 8.7;         // a double-precision floating-point constant
double sqrt(double);           // a function taking a double argument and
                               //  returning a double result
vector<Temperature> v;         // a vector variable of Temperatures (variable)
```

Declarations

- Declarations are frequently introduced into a program through “headers”
  - A header is a file containing declarations providing an interface to other parts of a program
- This allows for abstraction – you don’t have to know the details of a function or an object in order to use it. When you add

```
#include ”Matrix.h"
```

to your code, the declarations in the file `Matrix.h` become available

Definitions

A declaration that (also) fully specifies the entity declared is called a definition

- Examples

```
int a = 7;
int b;              // an (uninitialized) int
vector<double> v;   // an empty vector of doubles
double sqrt(double) { … };  // a function with a body
struct Point { int x; int y; };
```

- Examples of declarations that are not definitions

```
double sqrt(double);     // function body missing
struct Point;       // class members specified elsewhere
extern int a;       // extern means “not definition”
                    // “extern” is archaic; we will hardly use it
```

Declarations and definitions

- You can't *define* something twice
  - A definition says what something is
  - Examples

```
int a;          // definition
int a;          // error: double definition
double sqrt(double d) { … }  // definition
double sqrt(double d) { … }  // error: double definition
```

- You can *declare* something twice
  - A declaration says how something can be used

```
int a = 7;              // definition (also a declaration)
extern int a;           // declaration
double sqrt(double);         // declaration
double sqrt(double d) { … }  // definition (also a declaration)
```

Why both declarations and definitions?

- To refer to something, we need (only) its declaration
- Often we want the definition “elsewhere”
  - Later in a file
  - In another file
    - preferably written by someone else
- Declarations are used to specify interfaces
  - To your own code
  - To libraries
    - Libraries are key: we can’t write all ourselves, and wouldn’t want to
- In larger programs
  - Place all declarations in header files to ease sharing

Kinds of declarations

- The most interesting are
  - Variables
    - `int x;`
    - `vector<int> vi2 {1,2,3,4};`
  - Constants
    - `void f(const X&);`
    - `constexpr int s2 = sqrt(2);`
  - Functions
    - `double sqrt(double d) { /* … */ }`
  - Namespaces
  - Types (classes and enumerations)
  - Templates

Imperial College London

EXERCISE

- Create a main and write a function that prints the first and fourth columns of the table in slide 19.

Type Name	Kind of Value	Memory Used	Range of Values
`byte`	Integer	1 byte	−128 to 127
`short`	Integer	2 bytes	−32,768 to 32,767
`int`	Integer	4 bytes	−2,147,483,648 to 2,147,483,647
`long`	Integer	8 bytes	−9,223,372,036,8547,75,808 to 9,223,372,036,854,775,807
`float`	Floating-point	4 bytes	$\pm 3.40282347 \times 10^{+38}$ to $\pm 1.40239846 \times 10^{-45}$
`double`	Floating-point	8 bytes	$\pm 1.79769313486231570 \times 10^{+308}$ to $\pm 4.94065645841246544 \times 10^{-324}$
`char`	Single character (Unicode)	2 bytes	All Unicode values from 0 to 65,535
`boolean`		1 bit	True or false

- For example, for `int`, print:

```
#include <limits.h>
...
cerr << “\nINT min: ” << INT_MIN << “ ” << INT_MAX;
```

- Store the min and max in a variable. What happens if you sum 10 to the max or subtract 10 from the min?

EXERCISE

- Extend your main to create ten different declarations of variables. Mix as many types as you can.
- Print at least five of the different declared variables to screen.

Header Files and the Preprocessor

- A header is a file that holds declarations of functions, types, constants, and other program components.

- The construct

```
#include <iostream>
```

  is a "preprocessor directive" that adds declarations to your program
  - Typically, the header file is simply a text (source code) file

- A header gives you access to functions, types, etc. that you want to use in your programs.
  - Usually, you don't really care about how they are written.
  - The actual functions, types, etc. are defined in other source code files
    - Often as part of libraries

Imperial College London

Source files

- A header file (here, **Temperature.h**) defines an interface between user code and implementation code (usually in a library)
- The same `#include` declarations in both **.cpp** files (definitions and uses) ease consistency checking

Temperature.h:

```
// declarations:
class Temperature { … };
class Temperature_stream {
 Temperature get();
    …
};
extern Temperature_stream ts;
…
```

Temperature.cpp:

```
#include "Temperature.h"
//definitions:
Temperature Temperature_stream::get()
{ /* … */ }
Temperature_stream ts;
…
```

use.cpp:

```
#include "Temperature.h"
…
Temperature t = ts.get();
…
```

Scope

“When you declare a program element such as a class, function, or variable, its name can only be "seen" and used in certain parts of your program. The context in which a name is visible is called its *scope*.”

https://docs.microsoft.com/en-us/cpp/cpp/scope-visual-cpp?view=msvc-170

Types of scope:

Global scope

Namespace scope

Local scope

Class scope

Statement scope

…

Scope

- A scope is a region of program text
  - Global scope (outside any language construct)
  - Class scope (within a class)
  - Local scope (between { … } braces)
  - Statement scope (e.g. in a for-statement)
- A name in a scope can be seen from within its scope and within scopes nested within that scope
  - Only after the declaration of the name (“can’t look ahead” rule)
  - Class members can be used within the class before they are declared
- A scope keeps “things” local
  - Prevents my variables, functions, etc., from interfering with yours
  - Remember: real programs have **many** thousands of entities
  - Locality is good!
    - Keep names as local as possible

Scope

```
#include <vector>        // get max and abs from here
// no r, i, or v here
class My_vector {
    vector<int> v;            // v is in class scope
public:
    int largest()                  // largest is in class scope
    {
     int r = 0;               // r is local
     for (int i = 0; i<v.size(); ++i)  // i is in statement scope
          r = max(r,abs(v[i]));
     // no i here
     return r;
    }
    // no r here
};
// no v here
```

Scopes nest

```
int x;    // global variable – avoid those where you can
int y;    // another global variable

int f()
{
    int x;          // local variable (Note – now there are two x’s)
    x = 7;     // local x, not the global x
    {
     int x = y;     // another local x, initialized by the global y
               // (Now there are three x’s)
     ++x;      // increment the local x in this scope
    }
}

// avoid such complicated nesting and hiding: keep it simple!
```

Namespace

"Namespaces provide a method for preventing name conflicts in large projects."
https://en.cppreference.com/w/cpp/language/namespace

It is a way of 'naming' a scope.

To create a namespace you use:
namespace my_new_area {}

To refer to a namespace you use: my_new_area::Function_in_my_area();

To add a namespace to the global namespace you use:
using namespace my_new_area; //this should only happen in source files (cpp)

Recap: Why functions?

- Chop a program into manageable pieces
  - “divide and conquer”
- Match our understanding of the problem domain
  - Name logical operations
  - A function should do one thing well
- Functions make the program easier to read
- A function can be useful in many places in a program
- Ease testing, distribution of labor, and maintenance
- Keep functions small
  - Easier to understand, specify, and debug

Functions

- General form:
  - `return_type name (formal arguments);          // a declaration`
  - `return_type name (formal arguments) body      // a definition`
  - For example

```
double f(int a, double d) { return a*d; }
```

- Formal arguments are often called parameters
- If you don't want to return a value give **void** as the return type

```
void increase_power_to(int level);
```

  - Here, **void** means "doesn't return a value"

- A body is a block or a try block
  - For example

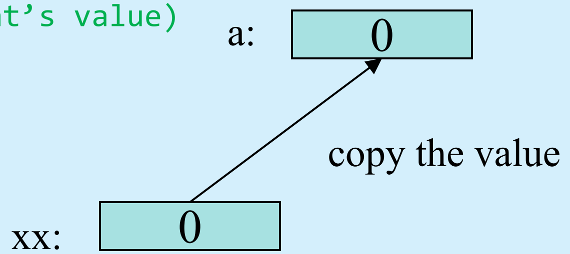
```
{ /* code */ }// a block
try { /* code */ } catch(exception& e) { /* code */ }     // a try block
```

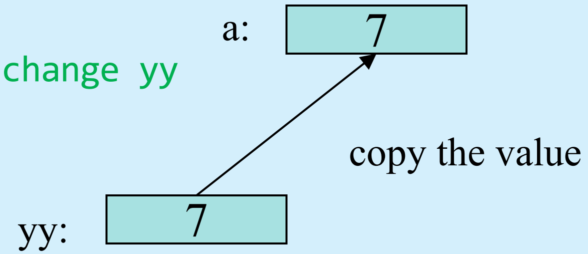
- Functions represent/implement computations/calculations

Functions: Call by Value

```
// call-by-value (send the function a copy of the argument's value)
int guu(int a) { a = a+1; return a; }


int main()
{
    int xx = 0;
    cout << guu(xx) << '\n';  // writes 1
    cout << xx << '\n';  // writes 0; guu() doesn't change xx
    int yy = 7;
    cout << guu(yy) << '\n'; // writes 8; guu() doesn't change yy
    cout << yy << '\n';  // writes 7
}
```



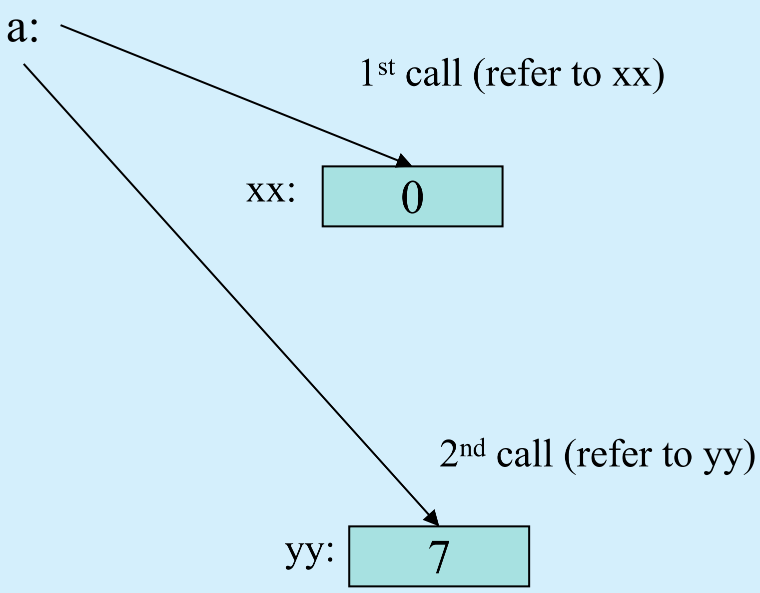


Functions: Call by Reference

```
// call-by-reference (pass a reference to the argument)
int guu(int& a) { a = a+1; return a; }


int main()
{
    int xx = 0;
    cout << guu(xx) << '\n';  // writes 1
                    // guu() changed the value of xx
    cout << xx << '\n';  // writes 1
    int yy = 7;
    cout << guu(yy) << '\n'; // writes 8
                    // guu() changes the value of yy
    cout << yy << '\n';  // writes 8

}
```




TIME FOR
A BREAK!

Imperial College London

Additional Slides

Functions

- Avoid (non-const) reference arguments when you can
  - They can lead to obscure bugs when you forget which arguments can be changed

```
int incr1(int a) { return a+1; }
void incr2(int& a) { ++a; }
int x = 7;
x = incr1(x);  // pretty obvious
incr2(x); // pretty obscure
```

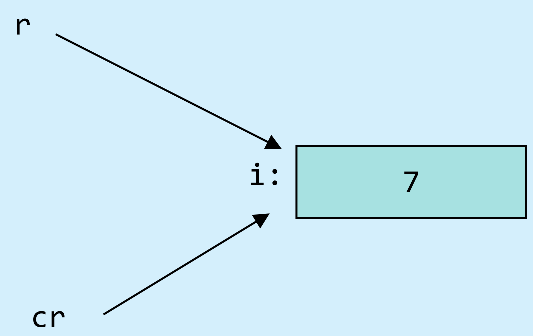
- So why have reference arguments?
  - Occasionally, they are essential
    - *E.g.*, for changing several values
    - For manipulating containers (*e.g.*, vector)
  - **const** reference arguments are very often useful

Call by value/by reference/ by const-reference

```
void f(int a, int& r, const int& cr) { ++a; ++r; ++cr; } // error: cr is const
void g(int a, int& r, const int& cr) { ++a; ++r; int x = cr; ++x; } // ok

int main()
{
    int x = 0;
    int y = 0;
    int z = 0;
    g(x,y,z);  // x==0; y==1; z==0
    g(1,2,3);  // error: reference argument r needs a variable to refer to
    g(1,y,3);  // ok: since cr is const we can pass “a temporary” (this is a tricky one!)
}
// const references are very useful for passing large objects
```

References



- “reference” is a general concept
  - Not just for call-by-reference

```
int i = 7;
int& r = i;
r = 9;    // i becomes 9
const int& cr = i;
// cr = 7;     // error: cr refers to const
i = 8;
cout << cr << endl; // write out the value of i (that’s 8)
```

- You can
  - think of a reference as an alternative name for an object
- You can’t
  - modify an object through a **const** reference
  - make a reference refer to another object after initialization

For example

- A range-for loop:

  - `for (string s : v) cout << s << "\n";` `// s is a copy of some v[i]`
  - `for (string& s : v) cout << s << "\n";` `// no copy`
  - `for (const string& s : v) cout << s << "\n";` `// and we don't modify v`

Imperial College London

Compile-time functions

- You can define functions that *can be* evaluated at compile time: **constexpr** functions

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

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
}
```

Imperial College London

Guidance for Passing Variables

- Use call-by-value for very small objects
- Use call-by-const-reference for large objects
- Use call-by-reference only when you have to
- Return a result rather than modify an object through a reference argument

- For example

```
class Image { /* objects are potentially huge */ };
void f(Image i);  … f(my_image);        // oops: this could be s-l-o-o-o-w
void f(Image& i); … f(my_image);        // no copy, but f() can modify my_image
void f(const Image&); … f(my_image);    // f() won't mess with my_image
Image make_image(); // most likely fast! ("move semantics" – later)
```

Imperial College London

Namespaces

- Consider this code from two programmers Jack and Jill

```
class Curve { /*…*/ };     // in Jack's header file jack.h
class Point { /*…*/ };     // also in jack.h

class Surface { /*…*/ };        // in Jill's header file  jill.h
class Point { /*…*/ };     // also in jill.h


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

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

Imperial College London

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 Curve{ /*…*/ };
     class Point{ /*…*/ };
  }

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

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

Imperial College London

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
- For example, **cout** is in namespace **std**, you could write:

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

Imperial College London

using Declarations and Directives

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

  you could write a "using declaration"
  - `using std::cout; // when I say cout, I mean std::cout`
  - `cout << "Please enter text… \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 text… \n";    // ok: std::cout`
  - `cin >> x;                    // ok: std::cin`

Imperial College London

EXERCISE

- Extend your main so that it has two loops:

- One `WHILE` loop, that progressively prints `INTEGER` numbers, by increasing the amount printed by an order of magnitude each time. Such as:

```
1
10
100
1000
```

- Now create a `FOR` loop, that progressively prints the same. What happens when numbers become very large? What happens if you use an `int` (INTEGER) vs a `float`, `double` or `long double`?