C++ Basics

- basic i/o: cout, cin
- comments
- variables: const, types, strong typing, casting
- operators: unary, binary
- Boolean expressions
- conditional: if, else, ?:
- loops: for, while
- arrays
- functions, call by value, prototypes, header files

basic i/o: cout

We start by learning the essential, non-Object Oriented features of C++.

No programming course is complete without the "Hello World" program:

```
#include <iostream.h>    // HelloWorld.cc

int main() {
    cout << "Hello World" << endl;
    return 0;
}
```

Anatomy of the program `HelloWorld.cc`:

1. `#include <iostream.h>`
   (a) `#include` is a C pre-processor directive (more on those later) to "include" the file `iostream.h`
   (b) the brackets `<...>` say that the file is in the "standard" include directory.
   (c) `iostream.h` (*much* more on that later) contains the standard C++ i/o class declarations and function prototypes (*much* more on that later).

2. `int main() {`
   (a) all C++ programs *must* have a `main`
   (b) `main` is actually a function (more on that later) of type `int`
   (c) `main` has no arguments (more on that later). Actually, `main` *can* have arguments – more on that later.
   (d) The body of `main` (and every function) is contained within { }.
   (e) C++ is written out free-form.
   How we write the code is a matter of style.

For space reasons on the slides, I will often break style guidelines.

3. `cout << "Hello World" << endl;`
   (a) cout is a pre-defined instance of the stream `ostream` – much more on that later. It is used for writing output to `tty`.
   (b) << is the *insertion* operator: it inserts what follows into the stream `cout`.
   (c) `"Hello World"` is the string we want to output.
   (d) `endl` does 2 things:
      i. it writes a <CR>
      ii. it flushes the output buffer
   (e) all statements in C++ must end with ‘;’

4. `return 0;`
   (a) since `main` is of type `int`, it must return an `int` to the program that called it (the shell).
   (b) a return value of ‘0’ signifies successful completion (that’s Unix, not C++).
5. }
   finally, we mark the end of the `main` code block.

We compile and link `HelloWorld.cc` with the command:

```
% g++ -o HelloWorld HelloWorld.cc
```

and run it with the command:

```
% ./HelloWorld
```

Notes:

1. C++ files can have many extensions, e.g:
   `.cc, .C, .cpp, .cxx`
   and probably others. Some are part of the standard, some are expected by certain compilers. We will use `.cc`, which is both standard, and works with g++.
2. The C++ version of gcc is invoked with “g++”. Since g++ and gcc are really the same beast, look at the *zillions* of command line options:

   ```
   % man gcc
   ```


3. What is the name of the executable file if we omit “`-o HelloWorld`”?
4. From now on, we will *always* use the compiler switch “`-Wall`” (= -**W**`arnings all`) to print all compiler warnings
5. We are compiling and linking together – we could do separately
6. Soon we will use “`make`” files – which are particularly useful for more complicated compilations.
7. For security, it is good to leave the current (working) directory out of the path – then we need to precede the executable name with “./”

more i/o: cin

- C++ is symmetric between *output* – with `cout`, and *input* – with `cin`.
- Before using `cin`, we have to jump ahead to variables.
  - All variables are *strongly typed* – a variable must be *declared* before it can be *defined*, or used.
  - C++ supports several *built-in* types: the number of bits used for each variable is implementation dependent. Since we've already seen `int` with `main`, we'll stay with `int`. On a 32-bit machine, `int` is usually a 32-bit signed integer.
- Let's write a program that reads in a number from the keyboard:


```
#include <iostream.h>   // ReadNumber.cc

int main() {
    cout << "Enter a number: " << ends;
    int i;
    cin >> i;
    cout << "You typed " << i << endl;
    return 0;
}
```


1. `cout << "Enter a number:  " << ends;`

   we don't want to use a <CR>, but we *do* need to flush the output buffer. This is done with `ends`.

2. `int i;`

   before we can use the integer `i`, we must *declare* it. The declaration can go *anywhere* in the same scope before `i` is used.

3. `cin >> i;`

   (a) the pre-defined input stream object is `cin`

   (b) the *extraction* operator is `>>`.

   (c) we extract the integer *from* the stream `cin` *into* the integer `i`.

   (d) when we type the <CR>, we automatically flush the stream.

4. `cout << "You typed " << i << endl;`

   we can use arbitrarily many `<<` (or `>>`) on the same line.

Comments

C++ supports 2 types of comment syntax:

1. A single line comment with: `//` (which can be *anywhere* on the line. E.g:

```
// now we're going to type a message
cout << "Enter a number: " << ends;
int i;      // i is an integer
```

2. The C-style "block" comment, `/* a comment */`
   This is useful for temporarily "commenting out" a block of code.

be careful using the 2 together, because // can comment out the /* or */.

Two comments about comments:

1. Style **Use comments liberally to document your code.**

   There are 2 reasons:

   (a) so you can understand your own code 24 hours later

   (b) so someone else (partner, TA, boss, successor) can understand it

2. Having said that, *well-written* C++ should be self-documenting.

- *At least* put a comment block at the beginning of each module, saying what that module does, who wrote it, etc.
- Document assumptions about parameters, validity, etc.
- Document the interfaces


```
   // Comments.cc
///////////////////////////////////////////////////////////
//
//    Copyright (c)  Michael Ogg, 1996
//    Author:  M. Ogg, ogg@ece.utexas.edu
//    Date:    Aug 21, 1996
//    Version: 1.0
//    Updated: Aug 21, 1996
//
//    Purpose:
//       This code does absolutely nothing.
//       But it's well documented
//
///////////////////////////////////////////////////////////
```


variables: types, strong typing, casting

The built-in types, and the *minimum* number of bits for a 32-bit architecture are:

type	bits	description
`char`	8	character
`short`	16	integer
`int`	32	integer
`long`	64	integer
`float`	32	floating point
`double`	64	floating point

All but `float` and `double` can be modified by `unsigned`

A variable must be *declared* before it is *defined* – but they can be done together. A variable can also be *initialized* with its declaration:

```
int i;
i = 1;
int j=i;
int k=2;
```

Variable Names:

- case sensitive
- alphanumeric, _
- begin with letter or _

operators: unary, binary

C++ supports the usual *binary* operators:

`+, -, *, /`

(binary, because there are *two* operands).

```
float a=2.0;
float b=5.0;
float c=6.0;
float arg2 = b*b - 4.0*a*c;
```

Operator precedence follows the usual *BODMAS* rules – when in doubt, use parentheses. C++ requires strong typing.

Some operators are used so frequently, there is a convenient shorthand:

Operator	Meaning
`a += b`	`a=a+b`
`a -= b`	`a=a-b`
`a *= b`	`a=a*b`
`a /= b`	`a=a/b`

Other miscellaneous binary operators:

Operator	Meaning
`a % b`	modulus of a/b
`a & b`	bit-wise AND
`a | b`	bit-wise OR
`a ^ b`	bit-wise X-OR

C++ supports *unary* operators too – there is only one operand.

Operator	Meaning	Comment
a++	a=a+1	postfix
++a	a=a+1	prefix
a--	a=a-1	postfix
--a	a=a-1	prefix
>>a		bit-wise Right shift
<<a		bit-wise Left shift
~a		1's complement


```
#include <iostream.h>   // Unary.cc

int main() {
    int i=4;
    cout << "i = " << i << ", i++ = " << i++ << endl;
    cout << "i = " << i << ", ++i = " << ++i << endl;
    return 0;
}
```


Casting

- Casting means converting one type to another. C++ does *not* enforce strong casting
- an expression with mixed types will *cast* one type to another – sometimes with unanticipated results.
- The unary operator `(type)` acting on a variable converts the variable to type `type`.
- When in doubt, use an explicit cast.


```
#include <iostream.h>   // Cast.cc

int main() {
    int i=4;
    int j=5;
    cout << "i/j = " << i/j << endl;
    cout << "i/(float)j = " << i/(float)j << endl;
    return 0;
}
```

const

- In C++, we use `const`, which creates a *run time* constant.
- A `const` cannot be altered once it is declared – so initialization (definition) must take place with declaration:

```
#include <iostream.h>   // Const.cc

int main() {
    const int i=42;
    const float pi=3.14159;  // also M_PI in math.h
    cout << "i = " << i << ", pi = " << pi << endl;
    return 0;
}
```


Boolean expressions

- A Boolean expression evaluates to either "true" (if *any* bit is set), or "false" (if *all* bits are zero).
- C++ supports a Boolean type, with values `true` and `false`.
- Boolean expressions are formed with Boolean operators:
- Parentheses should be used to resolve ambiguities in operator precedence.

Operator	Meaning
&&	logical AND
||	logical OR
!	logical NOT
==	equality
!=	inequality
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to


```
#include <iostream.h>   // Boolean.cc

int main() {
    int i=42;
    int j=137;
    cout << "i==j   " << (i==j) << endl;
    cout << "(i>j) || 1   " << ( (i>j) || 1) << endl;
    return 0;
}
```

conditional: if, else, ?:

Armed with the ability to form logical expressions, we can now do conditional execution.

i.e. execution conditional upon the truth of a Boolean variable or expression


```
#include <iostream.h>   // If.cc
#include <math.h>

int main() {
    cout << "Enter 3 numbers: " << ends;
    float a, b, c;
    cin >> a >> b >> c;
    float arg2 = b*b - 4.0*a*c;
    if ( (arg2>0.0) && (a!=0.0) ) {
        float arg = sqrt(arg2);
        cout << "Roots are: " << (-b + arg)/(2.0*a)
            << " and " << (-b - arg)/(2.0*a) << endl;
    }
    return 0;
}
```


Points to note:

1. `#include <math.h>` is used for math functions. We've sneakily introduced functions – more on them later.
2. `if { }` conditional expression. The Boolean expression is evaluated. If it resolves to `true`, the statements inside `{ }` are executed.
3. since C++ is written free form, statements can be written across several lines.
4. if only *one* statement follows the `if`, the { } are not necessary (tho recommended)

Very often, we not only want to execute statements after the `if`, but do something else if the Boolean is *not* true. This is done with `else`.


```
#include <iostream.h>   // IfElse.cc
#include <math.h>

int main() {
    cout << "Enter 3 numbers: " << ends;
    float a, b, c;
    cin >> a >> b >> c;
    float arg2 = b*b - 4.0*a*c;
    if ( (arg2>0.0) && (a!=0.0) ) {
        float arg = sqrt(arg2);
        cout << "Roots are: " << (-b + arg)/(2.0*a)
            << " and " << (-b - arg)/(2.0*a) << endl;
    }
    else
        cout << "I can't evaluate these roots" << endl;
    return 0;
}
```

Finally, the statement after an `else` can be another `if`:

```
#include <iostream.h>    // IfElseIf.cc

int main() {
    cout << "Enter a number: " << ends;
    int i;
    cin >> i;
    if      ( i<0 )
        cout << "number is < 0" << endl;
    else if ( i==0 )
        cout << "number is == 0" << endl;
    else
        cout << "number is > 0" << endl;
    return 0;
}
```


The construction:

```
if (condition)   do something
else             do something else
```

is used so often to evaluate an expression, that there is a shorthand operator:

```
a = (condition) ? b : c;
```

The condition is first evaluated.

- If it is true, `a` is set equal to the value of the expression `b`
- If it is false, `a` is set equal to the value of the expression `c`


```
#include <iostream.h>    // MinMax.cc

int main() {
    cout << "Enter 2 numbers: " << ends;
    float a,b;
    cin >> a >> b;
    cout << "the larger number is " << ( (a>b) ? a : b) << endl;
    return 0;
}
```


loops: for, while

To execute a block of code a number of times, or while some condition holds true. C++ provides the `for` and `while` loops.

```
#include <iostream.h>    // For.cc

int main() {
    cout << "Enter how many times to run loop: " << ends;
    int n;
    cin >> n;
    for (int i=0; i<n; i++) {
        cout << "i, i*i = " << i << ", " << i*i << endl;
    }
    return 0;
}
```

Points to Note:

1. The `for` expression has 3 parts, separated by ';'s
   (a) setting an initial value (`int i=0`).
   (b) a termination condition (`i<n`).
   (c) an action at the end of each iteration (`i++`).

   Any or all of these parts may be omitted, but the ";" is still necessary.
2. The code block to be executed is contained within `{ }`. If there is only 1 statement, the `{ }` can be omitted – but shouldn't be.
3. The `for` loop parameter, `i`, is valid only within the scope of the loop – within `{ }`

Sometimes, we want to execute a block of code while a condition holds true. This is done with a `while` loop.

```
#include <iostream.h>    // While.cc
#include <math.h>

int main() {
    float a=0.0;
    while ( a>=0.0 ) {
        cout << "sqrt(" << a << ") = " << sqrt(a) << endl;
        cout << "Enter a +ve number; -ve to end: " << ends;
        cin >> a;
    }
    return 0;
}
```


We could even have an endless loop (which is often useful). We might break out of the loop with ^C.

```
#include <iostream.h>    // Endless.cc

int main() {
    long i=0;
    while (1)
        cout << "This is the " << i++ << "'th iteration" << endl;
    return 0;
}
```

Note that since "1" has one bit set, it *always* evaluates to TRUE.

Sometimes we want a clean way of either skipping an iteration, or breaking out of a loop when some condition is met. This is done with the `continue` and `break` statements.

This smells awfully like `goto`, so should be avoided where possible

```
#include <iostream.h>    // ForContinue.cc

int main() {
    int n=50;
    for (int i=0; i<n; i++) {
        if ( !(i%7) ) continue;
        cout << "i, i*i = " << i << ", " << i*i << endl;
    }
    return 0;
}
```

```
#include <iostream.h>   // EndlessBreak.cc
#include <time.h>

int main() {
    long i=0;
    while (1) {
        cout << "This is the " << i++ << "'th iteration" << endl;
        if ( clock()>10 ) break;
    }
    return 0;
}
```

In this last example, we keep going indefinitely, until the used CPU time exceeds a certain number of ticks.

Clearly, there is no unique way to do what we want to do: the combination of `for, while, continue, break` is a matter of style.

arrays

Now that we can use loops, we can also use arrays.

```
#include <iostream.h>   // Array.cc
#include "stdlib.h"   // to fix SunOS
int main() {
    const kArraySize=10;
    float a[kArraySize], b[kArraySize];
    for (int i=0; i<kArraySize; i++) {
        a[i] = rand()/(float)RAND_MAX;
        b[i] = rand()/(float)RAND_MAX;
    }
    for (int i=0; i<kArraySize; i++) {
        cout << "element " << i << ": a, b, a*b "
            << a[i] << ", " << b[i] << ", " << a[i]*b[i] << endl;
    }
    return 0;
}
```


Notes:

1. To generate random numbers:
   (a) the header file `stdlib.h` is needed.
   (b) `rand()` is the random number generator that returns an `int`.
   (c) to convert to a float, in the range $0 \leq x \leq 1$, divide by `RAND_MAX`, using a cast.
2. a `const` is used for the array size. Its value is known at compile time, but it becomes a run time variable.
3. the operator `[]` is used to declare and access elements of the array.
4. the array's first element is `array[0]`
5. the index `i` is only valid within the scope of the `for` loop, so it can be "recycled" in subsequent loops.

multi-dimensional arrays

With the correct use of data structures, we actually use them much less than we'd think. But for some applications (e.g. matrices) they are still useful.

A multi-dimensional array is really an array of arrays:

```
float matrix[4][7];    // matrix[row][column]
```

i.e. `matrix` is an array of columns – the rightmost subscript changes the fastest.

functions, call by value, prototypes

We have sneakily used a few functions already.

- A function has a type:
  - `void` – no type
  - a built-in type, `int, float, long` etc.
  - a user-defined type (see later)
- A function returns a value, *unless* the function is of type `void`.
- A function can take zero, 1 or several arguments.
- The function arguments are passed *by value* from the calling program to the function. This means the function has its own *copy* of the parameters, and *does not change* the calling program's variables.

- The parameter names are only valid within the scope of the function (except for global parameters – avoid, but see later).
- Each function has a unique *signature* composed of:
  - the function's name
  - the function's class – see later
  - the function's argument types

  The function's return type is not part of the signature (see later for function overloading).
- A function must be:
  1. first declared (or prototyped)
  2. then defined (or implemented) – this can be done with declaration
  3. then invoked in the body of the code


```
#include <iostream.h>   // Function1.cc

void printMe(float x) {
    cout << "Number is: " << x << endl;
}

int main() {
    float a;
    while (1) {
        cout << "Enter a number: " << ends;
        cin >> a;
        printMe(a);
    }
    return 0;
}
```


Points to note:

1. The function `printMe` must be *declared* before its use.
2. The function can be *defined* at declaration time (but it doesn't have to be).
3. The parameter type(s) must be specified in the declaration. Note that `x` is a dummy parameter – any name would do, since it is local to `printMe`.
4. The function `printMe` is of type `void`.
5. Since it is void, there is *no* return value.
6. The argument passed to the function is `a`.
7. Since `printMe` is `void`, it is not used in an assignment statement.

What happens if `printme` is passed some variable that is *not* a float?

```
#include <iostream.h>   // Function2.cc
#include <stdlib.h>

void printMe(float x) {
    cout << "Number is: " << x << endl;
}

int main() {
    while (1) printMe(rand());
    return 0;
}
```

The absence of strong casting is a double-edged sword: it allows this to work, but may not always give the result we intended. (See later for function overloading and template functions).

We can see explicitly that the parameters really are passed by value:

```
#include <iostream.h>   // Function3.cc

int incrementMe(int x) { return ++x; }

int main() {
    int i=4;
    int j=incrementMe(i);
    cout << "i,j: " << i << ", " << j << endl;
    return 0;
}
```


Function Overloading

Since functions with a different *signature* are considered different *functions*, we can use this to *overload* function names:

```
#include <iostream.h>   // Function4.cc

float halveMe(float x) { return x/2.; }
int   halveMe(int   x) { return x/2; }

int main() {
    cout << "float halveMe: " << halveMe(5.0f) << ", "
        << "int halveMe: " << halveMe(5) << endl;
    return 0;
}
```


Notes:

1. The functions `float halveMe(float)` and `int halveMe(int)` really are 2 different functions. The linker decides which to use based on the signature.

2. `5.0` *without* `f` is a `double`, so the linker wouldn't know which function to use. We can either specify `5.0f` as a float, or cast 5 to a float with: `(float)5`

3. The return `type` is not part of the signature, so *cannot* be used to resolve ambiguities, since there is no strong casting.

Usually, we want to keep the function *definitions* in a separate file from their use. In this case, we must still *declare* the function by specifying its signature, or *function prototype*. This will usually be done in a header file. We will do this from now on.

```
#include "util.hh"   // Function5.cc

int main() {
    cout << "float halveMe: " << halveMe(5.0f) << ", "
        << "int halveMe: " << halveMe(5) << endl;
    return 0;
}
```


Notes:

1. We have put the function declarations in the file `util.hh` (we choose to use the `.hh` suffix to signify C++ header files).
2. Since `util.hh` is not a standard header file, it is enclosed in `"..."` not `<...>`
3. We have chosen to put `iostream.h` inside `util.hh` – since we know we'll always need it.

Let's look at `util.hh`

```
#ifndef __UTIL_HH   // util.hh
#define __UTIL_HH

#include <iostream.h>

float halveMe(float);
int   halveMe(int);

#endif // __UTIL_HH
```


Points to note:

1. `#ifndef __UTIL_HH`
   We only want to include the header file once, so we enclose it in an `#ifndef, #endif` block. This is a C pre-processor directive.
2. `#define __UTIL_HH`
   And then define a compile time variable to prevent subsequent inclusions.
3. `float halveMe(float);`
   The function prototype does not need to specify the actual parameter names – but it *must* specify the types. (That is the purpose).

Since we didn't put the function definitions in the header file, we'll define them in `util.cc`

```
#include "util.hh"   // util.cc

float halveMe(float x) { return x/2.; }
int   halveMe(int   x) { return x/2; }
```


Notes:

1. We also must include the same header file – even when it's not *technically* needed, it forces consistency between declarations and definitions.
2. Now we define the functions with the actual parameters.
3. If we change the function signature, we are forced to change *both* the header file *and* the implementation.
4. To build the executable, we can either compile both files together:

   ```
   % g++ -Wall -o Function5 Function5.cc util.cc
   ```

   or else first compile `util.cc` to make an object file, and then link:

   ```
   % g++ -Wall -c util.cc
   % g++ -Wall -o Function5 Function5.cc util.o
   ```


System Functions

C++ uses the standard C functions, as well as C++ ones. There are several families of functions, with their associated header files:

- "Standard" C functions. These are documented in e.g. K&R. The `include` files are usually in `/usr/include` or `/usr/local/include`. Note that many of these are made redundant or obsolete by C++. They are usually documented in the Unix `man` pages.
- "Standard" C++ functions. These are documented in e.g. E&S. The `include` files are usually (for gcc) in `/usr/include/g++` or `/usr/local/include/g++`. They will make more sense once we've covered more C++.

- "System" C functions. These are C functions specific to the Operating System. In the case of Unix, there will be a core set of "Posix-compliant" functions, plus additional OS specific functions. The "Posix-compliant" functions will often be defined on non-Posix systems, but it is not guaranteed. The `include` files are usually in `/usr/include` or `/usr/local/include`. They are usually documented in the Unix `man` pages.
- Library functions. These are C or C++ functions provided as part of a library. They may be used for e.g. graphics, database applications, etc.

Conclusion

You now know enough C++ to write pretty much any *non* Object Oriented program.