Types
and Operators

PHYS2G03

McMaster University

Operators

C++ has a large number of operators
http://www.cplusplus.com/doc/tutorial/operators

A large part of the C/C++ standard library is operators to do tasks that other languages do with functions

You can make new operators with objected oriented languages (e.g. C++, objective C, Fortran90)

Operations on Integers

- - negation a = -b;
- + Addition a = 2+b;
- - Subtraction a = 1-a;
- * Multiplication a = 2*b;
- / Division b = 2/3; = 0

Operations on Reals

- - negation a = -b;
- + Addition a = 2.0+b
- - Subtraction a = 1.1-a;
- * Multiplication a = 2.8*b;
- / Division b = 2.0/3.0; = 0.666667

Compound Operations

- Increment a=a+1; a++; ++a;
- Decrement a=a-1; a--; --a;

- + Addition a = a+b; a+=b;
- - Subtraction a = a-2; a-=2;
- * Multiplication a = a*b; a*=b;
- / Division a=a/b; a/=b;

Testing types

cp –r /home/2G03/types ~/

cd ~/types

make Shows a list of the programs

make compound

compound

You can look at any of them with an editor or more:

gedit compound.cpp &

more compound.cpp

Compound Operations

- Increment a=a+1; a++;
- Decrement a=a-1; a--;

Note: a++ is different to ++a if you re-use it

e.g. a=1; b=a++; gives b=1; a=2;
a=1; b=++a; gives b=2; a=2;

From /home/2G03/types
Try compound

Compound Operations

A C/C++ expression can have a value and a side effect

(a+=2) adds 2 to a and also has that value

You can do all sort of crazy things with C/C++

e.g. a=1; b= (a+=2)+2; // gives b=5

However, relying on strange side-effects is dangerous – don't do it!

Write code that is easy to understand

The compiler will generate efficient code – better to write clear C++! Short cryptic source code is not faster.

Complex Numbers

- Complex numbers are equivalent to two real numbers
- C++ offers complex with #include <complex>
- (C offers complex using <complex.h> but its different. You could use it but the C++ is cleaner.)
- You must specify the type to represent the real number, e.g. float

```
complex<float> z;
complex<double> zd;
```

Note: This is an example of a **template**

A type declared with <> in it where you can build it from other types. This is powerful idea not present in many languages.

Operations on Complex numbers

- \- negation a = -b;
- \+ Addition a = c+b;
- \- Subtraction a = b-a;
- * Multiplication by real a = 2.8f*b;

  by complex a = c*b;

Assignment: a = complex<float> (1,2)

Useful functions for Complex Numbers

abs(z)	Magnitude of complex number	
	$\text{sqrt}(x^2 + y^2)$	
z.real()	Real part	x
z.imag()	Imaginary part	y
conj(z)	Complex conjugate	
	$x - i\,y$	

Try: testcomplex program

Useful functions for Complex Numbers

abs(z)	Magnitude of complex sqrt($x^2 + y^2$)	
z.real()	Real part	x
z.imag()	Imaginary part	y
conj(z)	Complex conjugate x – i y	

Try: testcomplex program

C++ uses object oriented **polymorphism** concept
abs(complex) different to abs(float) etc….

Uses object oriented **class** concept
C++ complex is not just a type but also a class with associated functions like real and imag

Complex

```
#include <iostream>
#include <complex>
using std::complex;  // remove need to say std::complex
int main()
{
  complex<float> z;

  z = complex<float> (1.0,3.0);
  std::cout << "z complex = " << z << "\n";
  std::cout << "real z = " << z.real() << "\n";
  std::cout << "imag z = " << z.imag() << "\n";
  std::cout << "abs z = " << abs(z) << "\n";
  std::cout << "conj z = " << conj(z) << "\n";
  std::cout << "z * 5 = " << z*5.0f << "\n";
  std::cout << "z * (-1,2) = " << z + complex<float> (-1,2) << "\n";
}
```

Complex Numbers

- C++ complex is part of the standard library, not part of the main language and not as complete as e.g. Fortran 90 complex

```
complex<float> z;
z = z*5.0;      // Fails to compile!?
```

Because z uses floats and 5.0 is double by default

```
z = z*5.0f;     // works 5.0f is forced to float
```

Reason: complex<float> * float is defined

complex<float> * double isn't

Complex Numbers

- C++ complex is part of the standard library, not part of the main language and not as well supported as e.g. Fortran 90 complex

Reason: complex<float> * float is defined

complex<float> * double isn't

When you invent a new type you need to provide a function for every possible combination of variable types for every operator – g++'s compiler writers got lazy and skipped complex<float>*double

Text: Characters

C made text strings using arrays of characters

char a[] = “A string of text”;

You can also treat char as small integers;

a[0] = ‘A’; equivalently a[0] = 65;

(This is the ASCII standard for characters that just about every computer uses)

C provides functions for text like this when you #include <string.h>

Text: Characters C or C++

C made text strings using arrays of characters

char a[] = “A string of text”;

You can also treat char as small integers;

a[0] = ‘A’; equivalently a[0] = 65;

C provides functions for text like this when you #include <string.h>

C++ can use char as well, e.g.

std::cout << a << “\n”;

C++ string type

C++ offers a string type #include <string>

The string type has operators defined for it

```
string b = “a C++ string”,  c = “another one”, d;
d = b + c;  // join them          C version: strcat()
std::cout << d;  // output        C version: printf()
if (b<c) { // compare them }     C version: strcmp()
```

more string.cpp ; make string ; string

We will revisit this when we talk about formatting

Logical Types

In C any integer could be a logical type

0 == false anything else == true

In C++ the bool type (1 byte) was introduced

It can only be true 1 or false 0

The main purpose of bool is that it is clear that the coder intends it to be a logical result and it uses minimal storage (1 byte)

In practice it can still be used like an integer

Logical Operators

Most operators are binary, taking variables of two types and producing one of another type

e.g. + int + int == int

Comparison > float > float == bool

> int > int == bool

Comparison/Relational operators create a bool

Operators: > >= < <= == (equal) != (not equal)

Logical Operators for C++

There are also logical operators designed to operate on bool:

a && b is logical AND

a and b C++ only

only if both a and b are true, the result is true

a || b is logical OR a ^ b exclusive OR

a or b C++ only a xor b C++ only

if just one of a or b is true, the result is true

! is logical not (a unary operator) ! true == false

logic.cpp

make logic

logic

Note:
std::cout does not print “true” or “false”
but 1 for true
and 0 for false

Logical Operators for C

There are also logical operators designed to operate on bool:

a && b is logical AND

~~a and b~~

only if both a and b are true, the result is true

a || b is logical OR a ^ b exclusive OR

~~a or b~~ ~~a xor b~~

if just one of a or b is true, the result is true

! is logical not (a unary operator) ! true == false

logic.c

gcc logic.c -o logicc

logicc

Note: C only has && not "and" and || not "or".
They compile to the same thing.
C needs #include <stdbool.h> to use bool type

Bitwise operators

& is a bitwise AND – it is like && on every bit independently

binary 10010101 &

10110001 ==

10010001

| bitwise OR ^ exclusive OR ~ NOT

<< shift left >> shift right

13 << 2 == 52 like multiple by 2 twice

binary 1101 << 2 == 110100

Bitwise operators

& is a bitwise AND – it is like && on every bit independently

binary 10010101 &
10110001 ==
10010001

| bitwise OR ^ exclusive OR ~ NOT

<< shift left >> shift right

13 << 2 == 52 like multiply by 4

binary 1101 << 2 == 110100 (i.e. add 2 binary 0's)

binary 1101 >> 2 == 11 (decimal 3)

Operations

- Precedence: Some operations have priority over others:
- In order:
  – (negation), * and /, + and -

- What is A ? (code it up)
  A = 2*3/5+3*-4–3/4

Start with mytest.cpp, e.g.
gedit mytest.cpp &
make mytest
mytest

Operations

- Precedence based C/C++:

  int A;

  A = 2*3/5+3*-4-3/4

- Equivalent, more obvious C/C++:

  int A;

  A = ((2*3)/5) + (3*(-4)) - (3/4);

  = 6/5 - 12 - 0

  = -11

Use parentheses ( ) for clarity!

Results of mixed expressions

- C/C++ converts expressions when required. Typically it converts to the more complicated type (e.g. int + float = float)
- float x

  x = 1/2; // x = 0.0

  Conversion is at the last minute. 1/2=0 integer and then 0 is converted to float 0.0

- If you want a real number, include the decimal point

  x=1./2.; // x = 0.5

Type conversion: to float

- You can explicitly perform conversions to make sure the result is correct. In C/C++ this is known as a cast

```
float x;
x = float(1)/float(2);     // x = 0.5    C++ style cast

int i=1, j=2;
x = i/j;                          // x = 0.      no conversion
x = float(i)/float(j);      // x=0.5     C++ style
x = ((float) i)/((float) j);   // x = 0.5  C style (works in C++ too)
```

Type conversion: to integer

```
int i;
i = 1.6;   // default is round down, i=1
i = int(1.6);  // same effect

i = int(-3.4);   // i=???
std::cout << i;
```

(Try this yourself a bit within mytest.cpp)

Type conversion: to integer

```
int i;
i = 1.6;   // default is round down, i=1
i = int(1.6);  // same effect

i = int(-3.4);   // i=-3
std::cout << i;
```

Always rounds towards zero, not using normal math rounding rules (not 1.5 → 2 and 1.4 → 1)