# Variables and Types

PHYS2G03



McMaster University

# Variable Types

- Integer e.g. 1
- Real e.g. 2.34
- Complex e.g. -0.5 + 0.833 i
- Text e.g. “abc”
- Logical e.g. true

# Basic Variable Types

| | | |
|---|---|---|
| ■ Integer | int | e.g. 1 |
| ■ Real | float, double | e.g. 2.34 |
| ■ Complex | complex | e.g. -0.5 + 0.833 i |
| ■ Text | char, string | e.g. “abc” |
| ■ Logical | int, bool | e.g. true |

# More Variable Types

| | |
|---|---|
| ■ Integer | int, short, long |
| ■ Real | float, double |
| ■ Complex | complex<float>, complex<double> |
| ■ Text | char, string * |
| ■ Logical | int, bool |

Old C uses integers to represent true/false

0 is false, any non-zero value is true

* C++ only

# Testing types

cp –r /home/2G03/types ~/

cd ~/types

make Shows a list of the programs

make sizes

sizes

sizes uses the sizeof() function to determine how large in memory these types are in (bytes)

# The difference: size (bytes)

size of bool 1

size of char 1

size of short 2

size of int 4

size of float 4

size of double 8

size of complex<double> 16

# Initialization: Optional

```
int a = 1;
float x = 1.33333;
char greet[] = “hello”;
bool answer = false;

int a, b, c=1, d=2*5;
```

# Initialization

- Initialization is optional, if not done the variable will contain garbage until it is assigned a value (some compilers put 0)

```
int a = 1, b;
// a contains 1, b contains random data
…
b=2;
// a contains 1, b contains 2
```

# Constants

- Constants are not intended to be changed, this can be indicated to the compiler using the word **const** (safe programming, efficiency)
- Constants **MUST** be initialized when they are declared

```
const float G = 6.672e-8;  // cgs
const float k_B = 1.38066e-16;
const int N_particle_families = 3;
```

# Constants

- It's illegal to try to change the value of a constant in code (compiler error)

```
const int n_points = 20;


// Double the number of points
n_points = n_points*2;     ERROR
```

# Representing numbers on computers

High level languages define types of variables like int, float, etc…

In practice these are NOT quite the same as the mathematical concept

Computers have limitations in their ability to handle any integer or real number

# Number Representations

- Computers use discrete binary representation
- A Byte has 8 bits implies 256 possible values
- At *most* $256^b$ values can be represented by data using b bytes of storage
- e.g.

| | |
|---|---|
| 1 byte | 256 possible values |
| 2 bytes | 65,536 |
| 4 bytes | 4,294,967,296 |
| 8 bytes | 18,446,744,073,709,551,616 |

# Integer Representation

- Single byte:

  00000010 binary = decimal integer 2
- If the first bit is 1 then the number is negative (2’s complement representation)

  11111110 binary = decimal integer –2

If unsigned 11111110 = decimal 254 = 256-2

# Integer Representation

- 1 byte: -128 to 127 char
- 2 byte: -32768 to 32767 short
- **4 byte**: -2147483648 to 2147483647

  (typical default, e.g. **int a;** ) int
- 8 byte: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 long

- n byte: $-2^{8n-1}$ to $2^{8n-1}-1$

# Unsigned Integer Representation

- 1 byte:  0 to 255 unsigned char
- 2 byte:  0 to 65535 unsigned short
- **4 byte**:  0 to 4294967295 unsigned **int**
- 8 byte:  0 to 18,446,744,073,709,551,615 unsigned long

- n byte: 0 to $2^{8n}$-1

- Preface it with **unsigned** and get a factor of 2 bigger at the cost of no negative values

# Kilobytes, Megabytes, Gigabytes

On computers the normal definition of Kilo, Mega, Giga are often tweaked

- $2^{10}$ = 1024 ~ 1000 Kilo
- $2^{20}$ = 1048576 ~ 1000000 Mega
- $2^{30}$ = 1073741824 ~ 100000000 Giga

Unfortunately there is no consistent usage.

- RAM is typically 1024 = KB
- Disk is typically 1000 = KB

# Real Number Variables: Floating Point Representation

- Scientific Notation is handy way to represent a finite number of digits of precision and the magnitude of a real number (in powers of 10)

  e.g. $2.1x10^2 = 210$
- Not every real can be represented precisely (e.g.  surds like sqrt(3) or pi)
- Useful for very large or small numbers not suitable for integer representation

# Floating Point Representation

- To handle large/small values with limited precision computers offer floating point representation:

  $(+/-)M \times 2^E$
- A fixed number of bits are allocated to represent each part (M,E)
- Ordinary integers are fixed point: all bits are used to represent M and E is assumed to be 0
- So the Exponent E "floats" the decimal point to the right or left so that fractions and large numbers can be represented

# Floating Point Representation

e.g. 4 byte Floating Point (32 bits)

(typical default real, e.g. **float x**)

1 bit sign S, 23-bit mantissa M, 8 bit exponent E

- S = 0 positive, 1 negative
- Value = $(1+M/2^{23})2^{(E-127)}$
- Precision: 7 decimal digits
- Range: $10^{-38}$ to $10^{37}$

# Nasty numbers

- There are some expressions that are always too large to represent
- The CPU reserves some bit combinations to represent infinity (**inf, -inf**) and undefined values (**nan** “not a number”)

- It is best practice to detect such numbers
- Try program nasty which ignores bad values…

**make nasty**

**nasty**

No crash – shows bad values in prints

# nasty.cpp

```
#include <iostream>
#include <cmath>
int main()
{
  float r,s;

  s = 0.0;
  r = 1.0/s;
  std::cout << "1.0/0.0 = " << r << "\n";

  r = -1.0/s;
  std::cout << "-1.0/0.0 = " << r << "\n";

  r = 0.0/s;
  std::cout << "0.0/0.0 = " << r << "\n";

  r = sqrt(s-1.0);
  std::cout << "sqrt(-1.0) = " << r << "\n";
}
```

```
[~/types]$ nasty
1.0/0.0 = inf
-1.0/0.0 = -inf
0.0/0.0 = -nan
sqrt(-1.0) = -nan
```

sqrt(-1) is not a real number

# Floating point exceptions

## -ltrapfpe

Runtime errors are when a program does something illegal. You can ask the CPU to treat numbers too large to represent as errors.

On phys-ugrad you can force it to crash with **–ltrapfpe**

**c++ nasty.cpp –ltrapfpe –o nasty**

**nasty**

**Floating exception** now

-ltrapfpe uses special functions in **fenv.h**

such as **feenableexcept**

*see: /home/2G03/types/trapfpe.cpp*

Some compilers can also enable exceptions with options

**Without this math errors are ignored!**

**For your project use –ltrapfpe or your errors will be missed!**

# Testing real:

cp –r /home/2G03/types ~/

cd ~/types

make                Shows a list of the programs

make overflow

overflow

4 byte floating points can't get bigger than $10^{38}$

# overflow.cpp

```
#include <iostream>

int main()
{
  float r;

  r=1.0;
  for (;;) {
    std::cout << "r = " << r << "\n";
    r=r*2.0;
  }
}
```

no end test
Endless loop !

# Overflow

```
...
r = 6.33825e+29 = 633825300114114700748351602688.000000
r = 1.26765e+30 = 1267650600228229401496703205376.000000
r = 2.5353e+30 = 2535301200456458802993406410752.000000
r = 5.0706e+30 = 5070602400912917605986812821504.000000
r = 1.01412e+31 = 10141204801825835211973625643008.000000
r = 2.02824e+31 = 20282409603651670423947251286016.000000
r = 4.05648e+31 = 40564819207303340847894502572032.000000
r = 8.11296e+31 = 81129638414606681695789005144064.000000
r = 1.62259e+32 = 162259276829213363391578010288128.000000
r = 3.24519e+32 = 324518553658426726783156020576256.000000
r = 6.49037e+32 = 649037107316853453566312041152512.000000
r = 1.29807e+33 = 1298074214633706907132624082305024.000000
r = 2.59615e+33 = 2596148429267413814265248164610048.000000
r = 5.1923e+33 = 5192296858534827628530496329220096.000000
r = 1.03846e+34 = 10384593717069655257060992658440192.000000
r = 2.07692e+34 = 20769187434139310514121985316880384.000000
r = 4.15384e+34 = 41538374868278621028243970633760768.000000
r = 8.30767e+34 = 83076749736557242056487941267521536.000000
r = 1.66153e+35 = 166153499473114484112975882535043072.000000
r = 3.32307e+35 = 332306998946228968225951765070086144.000000
r = 6.64614e+35 = 664613997892457936451903530140172288.000000
r = 1.32923e+36 = 1329227995784915872903807060280344576.000000
r = 2.65846e+36 = 2658455991569831745807614120560689152.000000
r = 5.31691e+36 = 5316911983139663491615228241121378304.000000
r = 1.06338e+37 = 10633823966279326983230456482242756608.000000
r = 2.12676e+37 = 21267647932558653966460912964485513216.000000
r = 4.25353e+37 = 42535295865117307932921825928971026432.000000
r = 8.50706e+37 = 85070591730234615865843651857942052864.000000
r = 1.70141e+38 = 170141183460469231731687303715884105728.000000
Floating exception
```

Note – for a **float**
digits past the first 7 are usually not accurate so remember not to trust them.   Don't be fooled by the fact they are not zeroes

Here we are using powers of 2 – a special case of number floating point can represent

2^120 =
1 329 227 995 784 915 872 903 807 060 280 344 576
2^120+1 cannot be represented

Biggest number a float can do
$2^{127}$ = approx. $3 \times 10^{38}$

# Overflow and underflow

- When a number is too big to represent with that variable type, an **overflow error** occurs (crash)
- When a number is too small to represent it is an **underflow error**
- Underflows don't crash (by default), the variable is quietly set to zero

try: underflow program

Why does the value go below $10^{-38}$ before being zeroed? How is it representing those numbers?

# underflow.cpp

```
#include <iostream>

int main()
{
  float r;

  r=1.0;
  for (;;) {
    std::cout << "r= " << r << "\n";
    if (r == 0.0) break;
    r=r/2.0;
  }
}
```

float variable

endless loop

std::out to print current value for r each time

**break condition**
**r == 0** ?

# Double Precision

- Most machines also offer double precision:
  e.g. **double x**
  8 bytes or 64 bit floating point
  1 bit sign S, 52-bit mantissa M, 11 bit exponent E
- S = 0 positive, 1 negative
- Value = $(1+M/2^{52})2^{(E-1023)}$
- Precision: 15 decimal digits
- Range: $10^{-308}$ to $10^{308}$

# Extended precision

- Intel chips have a floating point unit with intrinsic 80 bit precision (better than double). Saving values to double (64 bit) loses precision!
- This way compiler optimization can change the answer
- Whether or not the hardware can do it, some languages also offer even higher precision, such as the long double. This might be slow if it is done by software.
- Intel CPU long double uses 80 bit precision (up to $10^{4931}$) and stores it in 128 bits (16 bytes) rather than (80 bits) 10 bytes. 80 bits is the best the hardware can actually do. You could do higher by hand (probably factor of 10-100 slower).

make overflow2

overflow2 | more

# What are the largest / smallest numbers?

Useful constants

FLT_MAX   largest float   #include <cfloat>

INT_MAX   largest int   #incude <climits>

Try: testprecision

# Precision in Practice

- Even though C/C++ will print out lots of decimal places when asked – in practice they are only meaningful up to a point
- **float** (4 byte): the first 7
- **double** (8 byte): the first 15

Try: precisionloss program
Convert the program to use double and see how much precision you can get

# precisionloss.cpp

```
#include <stdio.h>

int main()
{
  float r,s,add;

  add = 1.0;
  for (;;) {
    r=1.0;
    s = r+add;
    printf(" %25.20lf + %25.20lf = %25.20lf\n",r,add,s);
    add = add/2;
    if (r == s) break;
  }
}
```

float variables

endless loop

printf ‘C style’ text to terminal More compact that std::cout

**break condition s+add == s ?**

```
[wadsley@phys-ugrad types]$ precisionloss
  1.00000000000000000000 +   1.00000000000000000000 =   2.00000000000000000000
  1.00000000000000000000 +   0.50000000000000000000 =   1.50000000000000000000
  1.00000000000000000000 +   0.25000000000000000000 =   1.25000000000000000000
  1.00000000000000000000 +   0.12500000000000000000 =   1.12500000000000000000
  1.00000000000000000000 +   0.06250000000000000000 =   1.06250000000000000000
  1.00000000000000000000 +   0.03125000000000000000 =   1.03125000000000000000
  1.00000000000000000000 +   0.01562500000000000000 =   1.01562500000000000000
  1.00000000000000000000 +   0.00781250000000000000 =   1.00781250000000000000
  1.00000000000000000000 +   0.00390625000000000000 =   1.00390625000000000000
  1.00000000000000000000 +   0.00195312500000000000 =   1.00195312500000000000
  1.00000000000000000000 +   0.00097656250000000000 =   1.00097656250000000000
  1.00000000000000000000 +   0.00048828125000000000 =   1.00048828125000000000
  1.00000000000000000000 +   0.00024414062500000000 =   1.00024414062500000000
  1.00000000000000000000 +   0.00012207031250000000 =   1.00012207031250000000
  1.00000000000000000000 +   0.00006103515625000000 =   1.00006103515625000000
  1.00000000000000000000 +   0.00003051757812500000 =   1.00003051757812500000
  1.00000000000000000000 +   0.00001525878906250000 =   1.00001525878906250000
  1.00000000000000000000 +   0.00000762939453125000 =   1.00000762939453125000
  1.00000000000000000000 +   0.00000381469726562500 =   1.00000381469726562500
  1.00000000000000000000 +   0.00000190734863281250 =   1.00000190734863281250
  1.00000000000000000000 +   0.00000095367431640625 =   1.00000095367431640625
  1.00000000000000000000 +   0.00000047683715820312 =   1.00000047683715820312
  1.00000000000000000000 +   0.00000023841857910156 =   1.00000023841857910156
  1.00000000000000000000 +   0.00000011920928955078 =   1.00000011920928955078
  1.00000000000000000000 +   0.00000005960464477539 =   1.00000000000000000000
```

Junk digits

[wadsley@phys-ugrad types]$ pl2

1.00000000000000000000 + 1.00000000000000000000 =
2.00000000000000000000

00000001.000000000000000000000000 + 00000001.000000000000000000000000 =
00000010.000000000000000000000000

1.00000000000000000000 + 0.50000000000000000000 =
1.50000000000000000000

00000001.000000000000000000000000 + 00000000.100000000000000000000000 =
00000001.100000000000000000000000

**Extra digits created by print converting to decimal from binary. In binary they are just zeros**

...

$2^{-22}$

1.00000000000000000000 + 0.00000023841857910156 =
1.00000023841857910156

00000001.000000000000000000000000 + 00000000.000000000000000000000100 =
00000001.000000000000000000000100

$2^{-23}$

1.00000000000000000000 + 0.00000011920928955078 =
1.00000011920928955078

00000001.000000000000000000000000 + 00000000.000000000000000000000010 =
00000001.000000000000000000000010

$2^{-24}$

1.00000000000000000000 + 0.00000005960464477539 =
1.00000000000000000000

00000001.000000000000000000000000 + 00000000.000000000000000000000001 =
00000001.000000000000000000000000

# Scientific Computing Key issue: Numerical Accuracy

Loss of precision is also called round off error.

7 digits may seem like a lot, but if you add up

10 million numbers the error becomes huge!

Many standard computations require repeat operations that lead to roundoff

Try: roundoff

It sums 1 to n (a number you enter)

$1+2+3+\ldots+(n-1)+n = n\,(n+1)/2$

Try n=100, 10000, 100000, 1000000000

Where does each type start to fail?

Note that integers go funny for large values (see next slide)

# Roundoff

[wadsley@phys-ugrad ~/types]$ roundoff

Enter a number to sum the natural numbers to (e.g. 1000000)

1000*000*000

Sum of 1 to 1000000000 = 500000000500000000 (Exact)

float  18014398509481984.0000000000000000000

double 500000000067108992.0000000000000000000

int    -243309312

long   500000000500000000

For 1 billion added only the long type was able to keep up!

Note that both long and double have 64 bits of info – why did double fail and not long?

# Integer overflow

Integers don't go to inf if they get too big. They just wrap around to the other end

char a = 127;

a + 1 == -128

unsigned int a = 0;

a – 1 == 4294967295

Arian 5 explosion 1996
An integer overflow that cost 7 billion dollars
(Details in the audio or google it)

Try: intoverflow