Math and Pointers

Class 2

Questions

- questions about anything in Monday's material?
- questions about the assignment due tomorrow at noon?

Assignments

- you can re-submit as many times as you wish
- until the due time

- if you submit after it's due
  1. I may not see it unless you let me know
  2. it may get a late penalty

Powers of 2

know all the powers of 2 from 0 to 10

$$2^0 = 1$$
$$2^1 = 2$$
$$2^2 = 4$$
$$2^3 = 8$$
$$2^4 = 16$$
$$2^5 = 32$$
$$2^6 = 64$$
$$2^7 = 128$$
$$2^8 = 256$$
$$2^9 = 512$$
$$2^{10} = 1024$$

Powers of 2

- know the higher powers of 2
- know how they relate to powers of 10
- with these, you can interpolate

kibi	2^{10}	(1,024)	$\approx$	kilo	10^3	(one thousand)
mebi	2^{20}	(1,048,576)	$\approx$	mega	10^6	(one million)
gibi	2^{30}		$\approx$	giga	10^9	(one billion)
tebi	2^{40}		$\approx$	tera	10^{12}	(one trillion)
pebi	2^{50}		$\approx$	peta	10^{15}	(one quadrillion)
exbi	2^{60}		$\approx$	exa	10^{18}	(one quintillion)
zebi	2^{70}		$\approx$	zeta	10^{21}	(one sextillion)
yobi	2^{80}		$\approx$	yotta	10^{24}	(one septillion)

Example

about how much is 2^{37}?

Example

about how much is 2^{37}?

$$\begin{aligned} 2^{37} &= 2^{30} \times 2^7 \\ &\approx 10^9 \times 2^7 \\ &\approx 1 \text{ billion} \times 128 \\ &\approx 128,000,000,000 \end{aligned}$$

Exponents

in general adding a base raised to a power doesn't affect the exponent

$$x^a + x^a = 2(x^a)$$

Exponents

in general adding a base raised to a power doesn't affect the exponent

$$x^a + x^a = 2(x^a)$$

but there's a special case with a base of 2:

$$2^n + 2^n = 2^{n+1}$$

Logarithms

- $\log_a 1 = 0$
- $\log_a a = 1$
- $\log_a x^y = y \log_a x$
- $\log_a xy = \log_a x + \log_a y$ and similarly for division
- $a^{log_b x} = x^{log_b a}$
- $\log_a x = \frac{\log_b x}{\log_b a} = \log_a b \times \log_b x$

Logarithms

- logarithms are particularly important in algorithm analysis
- we deal almost exclusively with base-2 logarithms
- for our purposes, the simplest definition of a logarithm is

Logarithm

How many times can you divide a number n by 2
(using integer division)
before the result is 1 or 0?

- the answer to this question is the base-2 logarithm of n
- you can estimate base-2 logarithms directly from the powers of 2 table

Logarithm Notation

$\log n$	logarithm with arbitrary, non-specified base
$\log_{10} n$	base-10 logarithm when we need to specify
$\ln n$	natural logarithm
$\lg n$	base-2 logarithm

to typeset log in LaTeX use `\log`

Combinatorics and Summation

- the number of permutations of an n-element set is

$$P(n) = n!$$

- the number of k-combinations of an n-element set is

$$C(n, k) = \binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- the number of subsets of an n-element set is

$$2^n$$

- the sum of the first n positive integers is

$$\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Types

- a data type is
  - a set of values
  - a set of operations defined on those values
- in C++ (and most other languages) there are two types of types (sorry)
  - built-in primitive or fundamental types
  - programmer-defined aggregation types

Primitive Types

the main primitive types in C++ are

- void
- bool
- char
- short, int, long, long long
- float, double, long double
- char and the integer types can also have modifiers
  - signed
  - unsigned
- there are also pointers and references to these types

C++ Primitive Types

- because of historical decisions, the integer type system of C++ is a huge hot mess
- the two big problems are
  - unspecified sizes of the different types
    e.g., int and long int may be the same size, or may be different sizes — it's up to the compiler
  - totally screwy default decisions about signedness
    - rand() is guaranteed to return a non-negative result, but the return type is a signed integer
    - toupper('a') returns a signed int, not an unsigned char!
- therefore, safe and robust programming dictates very careful choices about types

Our Integer Types

we will use these types:

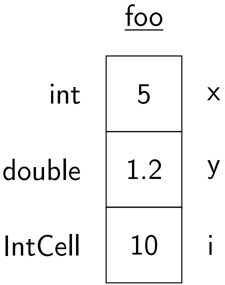
- unsigned for "normal" values that cannot be negative, which means counting and for loop control
- int for "normal" integer values that might be negative
- size_t for all size values including array and vector indices and container sizes and lengths
- specific-sized types when this is required, such uint64_t when we need the maximum counting capacity, defined in `<cstdint>`

- we will never use short or long or long long
- we will never use char as a numeric type, only for printable characters

Variables

- a variable is
    - a symbolic way to refer to a memory location
- when you declare a variable, space is allocated in the current local memory scope to hold one value of the variable's type
- the type of the variable indicates how to interpret the bits in that location

```
void foo
{
  int x = 5;
  double y = 1.2;
  IntCell i {10};
  ...
}
```



Immediate Variables

- variables like this have immediate storage
- the bits in the memory location are interpreted directly as that type

uint8_t	213	counter
int8_t	-43	my_value

the bits in each location are literally 1101 0101
how they are interpreted depends on the variable's type

Pointer Variables

- you can also declare a pointer variable using an asterisk
- we signify an uninitialized value with ?

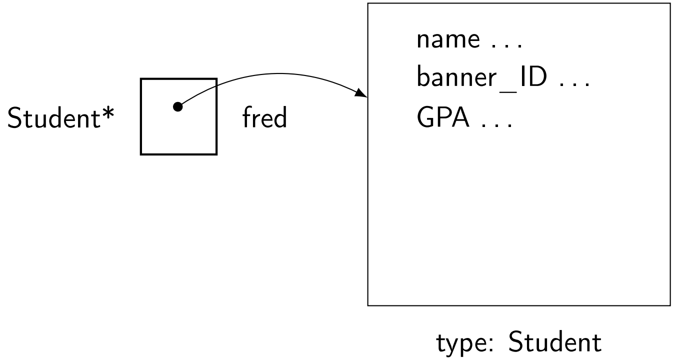
```
void foo()
{
  int x;
  double y = 1.2;
  IntCell i {10};
  int* p;
}
```

foo

int	?	x
double	1.2	y
IntCell	10	i
int*	?	p

Pointer Variables

- pointer variables have indirect storage
- the bits at that location represent an address at which to find a value of the declared type



Variable Types

in Java

- every primitive variable has immediate storage
- every object variable is a reference that points to someplace else

NO EXCEPTIONS!

Variable Types

in Java

- every primitive variable has immediate storage
- every object variable is a reference that points to someplace else

NO EXCEPTIONS!

but in C++

- a variable may be an immediate primitive or a pointer to a primitive
- a variable may be an immediate object or a pointer to an object

the programmer has to choose, every time a variable is declared

Declaring Variables

- an immediate variable is declared with a "plain" declaration (no asterisk)
- with or without an initialization

```
int foo;
double temperature = 12.345;
IntCell i {5};
```

- a pointer variable is declared with an asterisk

```
int* foo;
IntCell* i;
double* temperature;
```

but not `double* temperature = 12.345;`

Giving a Variable a Value

- the last example brings up the question:
- how does a pointer variable get its value?
  (the address to point to?)

Giving a Variable a Value

- the last example brings up the question:
- how does a pointer variable get its value?
  (the address to point to?)

- the same way any other variable gets a value: assignment

```
x = 5;
y = 3.72;
```

Getting An Address

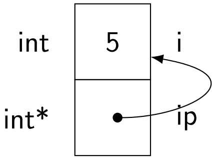
how do we get an address for a pointer variable?

- in C++ there are two ways to get the address of something to store in a pointer
- the first way is to use the address-of operator &

```
int i = 5;
int* ip;
ip = &i;
```

or more simply

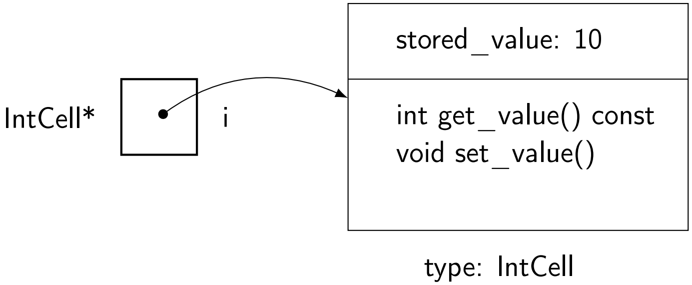
```
int i = 5;
int* ip = &i;
```



Getting An Address

- the second way is to use the return value of the new operator
- the new operator allocates a storage space from the heap for a variable and returns its address

```
IntCell* i = new IntCell {10};
```



Using a Pointer Variable

- once a pointer variable has a valid value, it can be used
- the value in the variable itself is an address, usually not directly useful
- to get at the value it's pointing to, we must dereference it using the dereference operator *

```
int i = 5;
int* ip = &i;

i++;
*ip += 5;
cout << "i is " << i << endl;
cout << "i is " << *ip << endl;
```

picture of memory from 2 slides ago

Using a Pointer to an Object

- regardless of whether the pointer points to an object or to a primitive, it works the same way
- however, the syntax presents a problem
  `*intcellptr.set_value(5);`
  does not work
- we must use the dereference-then-select operator ->
  `intcellptr->set_value(5);`

Memory Management

- when you allocate memory with new, that RAM location is unavailable for anything else in the computer
- when you're done with it, you must give it back
- failure to do so results in a memory leak
- memory allocated with new is released with delete
- C uses malloc and free, but not C++
  C++ uses new and delete
- there must be a 1-1 correspondence between calls to new and calls to delete in every program

Examples

code examples

- dereference.cpp
- segfault.cpp