New Types
(and objects)

PHYS2G03

McMaster University

New Types

cp –r /home/2G03/newtypes ~/

cd newtypes

ls

To make a program type

make name

(without the .cpp on the end)

Motivation for New Types

- Compound Data: Convenience

  e.g. a vector v1 has two or three real coordinate values

  using float or double is the obvious way to store them

  v1 = v2; more convenient than v1x = v2x; v2y=x2y;

- Clean Programming, consistency

  Using vector for all positions ensures you consistently make the same choice (e.g. all float) and code is easy to read and maintain

- Abstract Data Types: e.g. a set of cards

  How you store it isn't obvious but you know what you want it to do

  Leads naturally to *object oriented programming*

Making a New Type

- New types are built from existing types and are often compounds with multiple values.
- The new type needs to be defined for every piece of code that uses it (similar to function declarations)
- By default = operator defined for new type, but not other standard operators (e.g. no + for add etc…)
- Everything else is done with the standard types inside the new type
- Most languages now support new types,

  e.g. C, C++, Fortran 90, etc…

Note new types often called **objects**

C/C++ new types: **structures**

I can invent a 2d vector type:

```
struct vector { float x,y; };
```

The syntax is a name followed by a list of components inside braces. Any number of components is ok, e.g.

```
struct potluck { int a; double x,y; float b; };
```

C/C++ new types: **structures**

I can invent a 2d vector type:

```
struct vector { float x,y; };
```

Any code in the same block as this declaration can the use the new vector type.

If you put it at the top of a file – every function and other code can use the new type

C/C++ new types: **structures**

I can invent a 2d vector type:

```
struct vector { float x,y; };
```

You access the components by . and then their name. In the case of vector, x and y. e.g.

```
struct vector v;
v.x = 1+2.;
v.y = sin(1.57);
```

v.y is just like any other float variable

C/C++ new types: **structures**

I can invent a 2d vector type:

```
struct vector { float x,y; };
```

The new type can only use = (assignment)

Other standard operators don't work

```
struct vector v, w;
w = v;  // ok
v = 2*w;  // int * vector
            meaningless -- error
```

C++ new types: **struct**

In C++ the name of structure automatically refers to a structure of that type

typedef struct name { } name; //redundant;

```
struct vector { float x,y; };
struct circle { float x,y,radius; };

vector v;
circle mycircle;
radius a,b;

a=10.; v.x = a/sqrt(2.); v.y = a/sqrt(2.);
```

Making **circle** shorthand for **struct circle is** sensible
It is automatic in C++
(In C must use **typedef**)

C vs. C++ new types: **typedef**

```
struct thing { float a,b; };
typedef struct { float a,b; } thing2;
typedef struct thing3_struct { float a,b; } thing3;

  struct thing w;        // legal C/C++
  thing x;               // legal C++ only
  thing2 y;              // legal C/C++
  thing3 z;              // legal C/C++
```

try: make typedef ; make typedefc

C/C++ new types: **structures**

```
struct vector { float x,y; };
```

Structs with the same content are not equivalent, e.g.

```
struct coord2d { float x,y; };
```

This keeps them independent in case you change one. e.g. You could decide to use double in coord2d

try make equiv

structures

You can include any types you want:

e.g. struct planet { double mass; float x,y,z; };

e.g. struct person {

string name;

int age;

float height; };

Using **structs**

```
struct person {
  string name;
  int age;
  float height;  };

  person p;
  p.name = "Robert";    p.age = 42;    p.height = 172.0;

  std::cout << “The person’s name is “ << p.name << “, age is “
 << p.age  << “ and height is “ << p.height << “\n”;
```

C++: The Class

C++ introduced the idea of classes

class makes objects similar to struct – user defined data types

structs were retained for compatibility with C but can do everything a class can do

classes are designed to force object oriented programming

classes introduce the idea of public and private data and code (functions)   **… more later**

Object oriented programming

Hierarchy of **structs (C++ only)**

```
struct person {
  string name;
  int age;
  float height; };
struct employee : person {
  string title;
  float salary;
};
employee em;
  person p2;

  p2.name = "Robert";
  p2.age = 42;
  p2.height = 172.0;

  ((person&) em) = p2;
  em.title = "foreman";
  em.salary = 50000.;

  std::cout << "The person who is " << em.title
        << " is named " << em.name << "\n";
```

employee struct inherits the properties of person

person is the base
employee may be treated as a person using a cast

Only one . needed now
em.name

Object oriented programming

Hierarchy of **classes (C++ only)**

```
class person {  public:
  string name;
  int age;
  float height; };
class employee : public person {  public:
  string title;
  float salary;
};
  employee em;
  person p2;

  p2.name = "Robert";
  p2.age = 42;
  p2.height = 172.0;

  ((person&) em) = p2;
  em.title = "foreman";
  em.salary = 50000.;

  std::cout << "The person who is " << em.title
        << " is named " << em.name << "\n";
```

employee class inherits the properties of person class

person is the base
employee is a subclass of person
employee may be treated as a person using a cast

Object oriented programming

Hierarchy of **classes (C++ only)**

```
class person {  public:
  string name;
  int age;
  float height; };
class employee : public person {  public:
  string title;
  float salary;
};
  employee em;
  person p2;

  p2.name = "Robert";
  p2.age = 42;
  p2.height = 172.0;

  ((person&) em) = p2;
  em.title = "foreman";
  em.salary = 50000.;

  std::cout << "The person who is " << em.tit
       << " is named " << em.name << "\n”;
```

Note the use of **public** everywhere.

Stroustrup’s idea of object oriented programming (C++) emphasized hiding data so he made it the default

I added the word **public** to make the data visible again so I can use the data directly

Object oriented programming

Hierarchy of **classes (C++ only)**

```
class person {
  string name;
  int age;
  float height;
public:
  void setname( string newname ) { name = newname; };
  void setage( int newage ) { age = newage; };
  void setheight( float newheight ) { height = newheight; };
};

  person p2;
  p2.setname( "Robert" );
  p2.setage( 42 );
  p2.setheight( 172.0 );
```

outside access to variables in classes (by default) is via functions only

Note: void functions don’t return anything back but something happens – data was set in the class

Example files:

employee_structc	C style struct
employee_struct	C++ style struct (inheritance)
employee_class	C++ public class (direct access to data)
employee_classprivate	C++ private class (C++ default) (Long!)

Class vs. Struct

In C++ **struct** and **class** are 99% equivalent

The main difference is that classes make all their contents to be private by default

In the example the code explicitly overrode that to allow main to see the contents of person and employee (making struct and class equivalent)

Encapsulation (also called **Data hiding):**

Private data is a key idea in object oriented programming. We will discuss that more later.

For now we will use struct and keep everything visible.

Data Hiding

The Stroustrup’s idea of always hiding data is very laborious for applications like scientific computing.

It requires a lot of work making “setter and getter” functions which slow down the code a lot without making it easier to read or use.

Scientific code libraries tend to avoid this approach for math heavy code

e.g. **eigen** C++ library for linear algebra
http://eigen.tuxfamily.org

Functions and struct

A key advantage of new types is keeping function argument lists short and simple

```
addperson( string name,  int age, float height );
vs.
struct person {  string name;   int age;  float height; };
// more work once only
addperson(  person p );  // cleaner
```

Functions and struct

A second key advantage is minimizing changes when new variables are required

addperson(char *name, int age, float height, float weight);
// Every use of this function must change to add new argument
vs.
struct person { char name[30];   int age; float height, weight }; // only change required

addperson(person p); // no changes needed
calculateBMI(person p);
calculateRetirement(person p, int year);

Functions and struct

You should aim to keep struct info in one place:

Put them in a header file (can be same file with prototypes of functions that use it)

e.g. A file called **person.h**:

```
struct person {  string name;   int age;  float
height, weight;  };
// function prototypes
void addperson( person p );
```

Example: Vector Algebra

Consider a vector type with 3 real numbers

e.g. struct vector { float x,y,z; };

Consider basic vector algebra on vectors a and b

- dot product   a.b
- vector magnitude |a| = sqrt(a.a)
- angle   cos(theta) = a.b/|a|/|b|
- cross product   a x b

Vector Algebra C/C++

```
typedef struct { float x,y,z; } vector;

float dot( vector a, vector b) {
  return (a.x*b.x + a.y*b.y + a.z*b.z);
}

float magnitude( vector a ) {
  return sqrt(dot( a, a ));
}

vector cross( vector a, vector b) {
  vector c;
  c.x = a.y*b.z - a.z*b.y;
  c.y = a.z*b.x - a.x*b.z;
  c.z = a.x*b.y - a.y*b.x;
  return c;
}
```

definition of vector C style

Vector Algebra C++

```
struct vector { float x,y,z; };

float dot( vector a, vector b) {
  return (a.x*b.x + a.y*b.y + a.z*b.z);
}

float magnitude( vector a ) {
  return sqrt(dot( a, a ));
}

vector cross( vector a, vector b) {
  vector c;
  c.x = a.y*b.z - a.z*b.y;
  c.y = a.z*b.x - a.x*b.z;
  c.z = a.x*b.y - a.y*b.x;
  return c;
}
```

definition of vector C++ style

functions using vectors

Note: This function returns a vector type. Structures are a good way to return more than one value.

A vector type library

The definition of vector and prototypes of the functions should go in a file “vector.h”

Then any program using vectors can

#include “vector.h” and know how to use the type and the functions

The functions make a self contained little library we can keep in a file “vector.cpp”. If we compile it to vector.o we can use those functions by linking it to any program that wants them, e.g.

make testvector; testvector

Vector Algebra

vector.h
C style

```
typedef struct { float x,y,z; } vector;

float dot( vector a, vector b);
float magnitude( vector a );
vector cross( vector a, vector b);
```

Vector Algebra

testvector.cpp

```
#include <iostream>
#include <cmath>
#include "vector.h"

int main()
{
  vector v1,v2,v3;

  std::cout << "Enter 3 components of vector 1\n";
  std::cin >> v1.x >> v1.y >> v1.z;
  std::cout << "Enter 3 components of vector 2\n";
  std::cin >> v2.x >> v2.y >> v2.z;

  std::cout << "The dot product is " << dot(v1,v2) << "\n";
  std::cout << "The angle is " <<
180/M_PI*acos(dot(v1,v2)/magnitude(v1)/magnitude(v2)) << " degrees\n";
  v3 = cross(v1,v2);
  std::cout << "The cross product is " << v3.x << " " << v3.y << " " << v3.z << "\n";
}
```

Vector Algebra

Makefile

```
testvector: testvector.o vector.o
	c++ testvector.o vector.o -o testvector

vector.o: vector.cpp vector.h Makefile
	c++ vector.cpp -c

testvector.o: testvector.cpp vector.h Makefile
	c++ testvector.cpp –c
```

A vector object

Object oriented languages (C++) offer convenient features for new types such as **operator overloading**: The ability to define what +, - and any other math operators (dot product) mean for your new types. This makes for cleaner looking code:

```
c = a+b;
```

Replaces

```
c = add_vector(a,b);
```

But it is functionally the same (same machine code)

If your program does a lot of vector operations it will be easier to write, read and re-use.

Vector Algebra

vector.h

```
struct vector { float x,y,z; };


float dot( vector , vector );
float magnitude( vector  );
vector cross( vector , vector );
operator+(vector, vector );
```

C++new operator

variable names not necessary in prototype – just the type

New + operator

```
vector operator+(vector a, vector b) {
  vector sum;
  sum.x = a.x+b.x;
  sum.y = a.y+b.y;
  sum.z = a.z+b.z;
  return sum;
}
```

vector.cpp

C++ only

```
#include "vector.h"
int main()
{
  vector v1,v2,v3;
  std::cout << "Enter 3 components of vector 1\n";
  std::cin >> v1.x >> v1.y >> v1.z;
  std::cout << "Enter 3 components of vector 2\n";
  std::cin >> v2.x >> v2.y >> v2.z;

  v3 = v1+v2;
  std::cout << "The sum of v1+v2 is " << v3.x << " " << v3.y << " " << v3.z << "\n";
}
```

testvector.cpp

C++ Standard objects

C++ has a lot of pre-made objects (classes)

e.g. std::string, std::cout, std::cin, std::complex

You can avoid writing std:: all the time by “using namespace std;”.

This tells the compiler you want it to assume any standard name is ok to use without having “std::” in front of it, e.g.

```
#include <string>
#include <iostream>
using namespace std;

int main() {
  string a=“hello”;
  cout << a << “\n”;
}
```

Standard class: std::cout

cout overloaded << to call a print function when given ordinary types.

It needs a separate functions for print int (integer), print float, etc… which is how it knows what to do for each type, e.g.

x=2;

cout << “x equals “ << x;

Does print characters, then print int

See e.g.

http://www.cplusplus.com/reference/iostream/cout

Standard class: std::cout

cout overloaded << to do other tasks like set private internal data about formats to use

#include <iostream>

#include <iomanip>

std::cout << std::setprecision(20)

Sets an internal variable to make future prints have 20 digits after the decimal place (see pi.cpp)

See e.g.

http://www.cplusplus.com/reference/iostream/cout

Standard class: std::cout

std::cout does not overload << to be aware of new types that you make. If you want to print your new types – you need to write your own print functions for them

myawesomeclass x;

std::cout << "My class x type equals " << x;

Will not compile – no idea what to do with a cout combined with an x.

Standard class: std::string

string has overloaded operators such as + to use it to combine strings, > to compare strings and so on.

It also includes functions attached to the object to query the string, e.g.

string a=“hello”;

cout << “a is “ << string.length() << “ letters long”;

See e.g.

http://www.cplusplus.com/reference/string/

Standard class: std::complex

It is a special kind of object based on template. The types it is made from can be specified in your program.

complex<float> z;

complex<double> dz;

Complex has operators such as + (add), * (multiply), etc

Functions in the class, e.g. z.imag() for imaginary part

And new versions of math functions that are able to handle the new type as well, e.g. exp(z) which returns a complex value

All these are equivalent to doing it by hand with two real numbers, e.g.

float x,y;

However, it saves you work and makes for clearer readable code to use the standard complex class.

Note that there is no data hiding going on!

See e.g. http://www.cplusplus.com/reference/complex

Standard class: std::vector

The standard template library has a vector type

#include <vector>

It is an object based on template. The types it is made from can be specified in your program.

vector<float> v(10);   // vector with 10 elements

It is not for vector algebra – no math operations are defined for it.   It is just a special kind of variable length list.

See e.g.
http://www.cplusplus.com/reference/vector/vector/