Intro to C++

- C++

References

- Alternative name for an object (pointers in disguise)

```
void f()
{
    int i = 1;
    int& r = i;
    int x = r;       // x = 1
    r = 2;           // i = 2
}
```

Main usage of references

- Arguments to functions (especially const)
  - int length(const string& s)

```
// bad style
void increment(int& a) {a++;}

void f() {
  int x = 1;
  increment(x);
}
```

```
// better style – argument clearly
modified
int next(int p) {return p+1; }
void incr(int* p) {( *p) ++; }
void g() {
  int x = 1;
  increment(x);
  x = next(x);
  incr(&x); }
```

Pointer to void

- Pointer to ANY type of object
  - can be compared but not manipulated

```
void f(int *p)
{ void* pv = pi;  // ok – implicit conversion
   *pv;              // error can't dereference void *
   pv++;             // error can't increment void *(size of object unknow
   int* pi2 = static_cast<int *> (pv);  // explicit conversion
   double* pd1 = pv;    // error
   double* pd2 = pi;    // error
   double* pd3 = static_cast<double*> (pv);     // unsafe
```

Structures

- Aggregate of related types

```
struct adress {
    char* name;
    long int number;
    char* street;
    char* town;
    char state[2];
    long zip;
};
```

New type called address

! Semicolon necessary after curly brace

Usage

```
void f()
{address id;
   id.name = “George”;
   id.number = 61;
}
```

```
void print_addr(address* p)
     cout << p->name << “\n”
           << p->number << “ “
           << p->street << “\n”
```

p->m is equivalent to (*p).m

Objects of structure types can be assigned, passed as function argumen
and returned as results

```
address current;
address set_current(address next)
{ address prev = current;
   current = next;
   return prev;
}
```

Name of structures

- The name of a type becomes available for immediate use after it has been encountered

```
struct Link {
    Link* previous;
    Link* successor;
}
```

However not possible to declare new objects until complete definition

```
struct No_good {
    No_good member; // error
};
```

Declarations / Definitions

- An object must be defined exactly once in a program. It may be declared many times but the types must agree exactly.

Some more stuff

```
// structure refering to each other
struct List;   // to be defined later

struct Link {
    Link* pre;
    Link* suc;
    List* member_of;
};

struct List {
    Link* head;
};
```

```
struct S1 {int a};
struct S2 {int a};
```

are two different types
(name equivalence)

```
S1 x;
S2 y = x; // error
```

EVERY structure has
a UNIQUE DEFINITION
in a program

Functions

- Function declaration
  - name, type of returned value, arguments
  - Elem* next_elem(); void exit(int);
- Function definition
  - Function declaration + body

```
void swap(int* p, int* q)
{int t = *p;
  *p = *q;
  *q = t;
}
```

Pass-by-value, Pass by reference

```
void f(int val, int& ref)
{
  val++;
  ref++;
}
```

Passing by reference for efficiency reasons

```
void f(const Large& arg)
{

}
```

Default arguments

```
void print(int value, int base = 10);    // default base is 10

void f()
{
  print(31);
  print(31, 10);
  print(31, 16);
}
```

Pointers to Function

- Things you can do a function
  - call it
  - take it's address

```
void error(string s) { ...... }

void (*efct) (string);

void f()
{
  efct = &error;
  efct = error;
  efct(“foo”);
  (*efct)(“foo”);
}
```

Classes

The aim of the C++ class concept is to provide the programmer with a tool for creating new types that can be used as conveniently as the built-in types. In addition, derived class and templates provide ways to organize related classes that allow the programmer to take advantage of their relations.

B. Stroustrup

Encapsulation

```
// without encapsulation
struct Date {int d,m,y;}

void init_date(Date &d, int, int, int);
void add_year(Date& d, int n);
```

```
// witt encapsulation
struct Date {int d,m,y;

void init_date(Date &d, int, int, int
void add_year(Date& d, int n);
}
```

Functions within a class definition are called members.
A struct is a class with all member public

Access restriction

```
class Date {
    int d, m, y;
public:
    void init(int d, int mm, int yy);
    void add_day(int n);
};
```

private part only accessible through members

public can also be accessed from the outside world

Constructors

- Ensure objects initialized once

```
class Date {
    int d, m, y;
public:
    Date(int, int, int);
    Date(int, int);
    Date(int);
    Date();
    Date(const char *);
};
```

same name as the class

```
Date today(4);
Date july4(“July 4, 1983”);
```

# of constructors can be reduced using default arguments

Const member functions

```
class Date {
      int d, m,y;
public:
      int day() const {return d; }
      int moth() const {return m;}

};
```

These functions do not modify the state of a Date

Self reference

```
d.add_day(1).add_month(1)
```

Object must return a reference to itself

```
Data& Date::add_year(int n)
{
    .....

    return *this;
}
```

this is a special variable which you can think of as a pointer to the object

Destructors

- Free resources
  - Automatically called
  - Variable goes out of scope
  - Delete object in free store
  - constructor/destructors

```
class Name {const char *s};

class Table {
    Name*p;
    size_t sz;
public:
    Table(size_t s = 15)
   {p = new Name[sz=s]; }
   ~Table() {delete [] p; }
   Name* lookup(const char *);
   bool insert(Name *);
}
```

Construction and Destruction

- Constructor of local variable executed each time the thread of control passes through the declaration
- Destructor executed each time the local variable's block is exited
- Destructors for local variables are executed in reverse order of their construction