C++ Basics

Example class: `Expr`

- **class definition**:

```
class Expr {
  /* constructor */
  Expr() { ... }

  /* destructor */
  virtual ˜Expr();

  /* copy constructor */
  Expr(const Expr& rhs) {
    field = rhs.field();
  }

  /* copy assignment operator */
  Expr& operator=(const Expr& rhs) {
    if (this == &rhs) /* check self-asgn */
      return *this;
    field = rhs.field();
    return *this;
  }

  /* constant member function */
  int foo() const {
    /* no object state change */
  }
};
```

- **object initialization**:

```
Expr e1;                  /* constructor */
Expr e2(e1);         /* copy constructor */
Expr e3 = e1; /* copy assignment opreator */
```

Uses of constructors

- **Initialization** (giving objects its first value) of objects generated by *structs* and *classes* is performed by **constructors**.
- **default constructor**: one that can be called with without any arguments:

```
class A {
  A();
}
class B {
  explicit B(int x = 0);
}
```

- there are multiple types of constructors:
- **copy constructor**: used to **initialize and construct** an object with a different object of the same type
- **copy assignment operator**: used to **copy** the value from one object to another of the same type

Operator overloading

```
class Expr {
  Expr* operator&();
  Expr operator++(int);
};
```

Functors Functor is a special object which acts as a function.

```
struct add_x {
  int x;
  add_x(int x) : x(x) {}
  int operator()(int y) { return x + y; }
};

add_x add42(42);
int i = add42(8);  // returns 42 + 8
```

Overloaded functions

- Two functions can have **same function name** with different signatures.

Function pointers

- **function that takes a function pointer as an argument**

```
int foo(int x, int (*funarg)(int, int));
```

- **using to funptr vars**

```
int add(int a, int b) { return a + b; }
int (*sum)(int, int) = add;
... foo(10, add) ...
```

Structure definition

```
struct tree_t {
  int value;
  struct tree_t *left;
  struct tree_t *right;
};
typedef struct tree_t bintree_t, *bintree_p;
```

Virtual functions

- **virtual function**: runtime automatically invokes the proper member function when it is overridden by a derived class
- **pure virtual function**: `virtual void foo() = 0`; derived class *must* define the function.
- **virtual destructor**: always make classes with virtual functions contain virtual destructor; this will ensure that correct destructor will be invoked

```
class Base {
  ˜Base throw(); /* non-virtual */
};
class Derived : public Base {
};
void wrongFunc(Base *b) {
  /* only fields related to base is removed */
  delete base;
}
...
  Base *base = new Derived();
  wrongFunc(base);
...
```

Use consts

```
/* READ BACKWARDS! */
/* p is a constant pointer to constant char */
char greeting[] = "Hello";
const char * const p = greeting;

/* does not modify the object */
char& Stream::getChar() const;
```

References vs pointers

- reference must be initialized when it's created
- once a reference is initialized to an object, it cannot be changed to refer to another object
- there is no "NULL" reference
- (-) pointer arithmetic not possible
- (+) no dereference needed

Argument passing: use pass-by-const-reference

- in C where only call-by-value was available, we needed to pass-by-"pointer"
- now, call-by-reference in C++ is just as efficient (no copy-in, copy-out as in call-by-value, which involve constructor/destructor call)s and it's safer

Template specialization

```
template<typename T>
class A {
  T element;
  foo(T arg) { T.inc(); }
}
/* template specialization */
template<>
class A <int> {
  int element;
  foo(int arg) { arg++; }
}
```

Type casting

- **dynamic_cast**: between pointers/references to objects; successfully only casted to its base type (**upcast**); *runtime-checking*
- **static_cast**: between (related) pointer types (can be used for **downcast**)
- **reinterpret_cast**: between any (possibly unrelated) pointer types

Smart pointers: auto_ptr

- deprecated; use **unique_ptr** instead
- `#include <memory>`

```
template <class Y>
struct auto_ptr_ref {};

template<class X>
class auto_ptr {
public:
  typedef X element_type;
  explicit auto_ptr(X* p = 0);
           auto_ptr(auto_ptr&);
  template<class Y> auto_ptr(auto_ptr<Y>&);

  auto_ptr& operator=(auto_ptr&);
  template <class Y> auto_ptr& operator=(auto_ptr<Y>&);
};
```

- **usage**:

```
void f() {
  auto_ptr<int> pt(new int);
  /* get pointer */
  ... pt.get() ...
}
```