Recuperación de Información Multimedia

C++ y OpenCV

CC5213 – Recuperación de Información Multimedia
Departamento de Ciencias de la Computación
Universidad de Chile
Juan Manuel Barrios – https://juan.cl/mir/ – 2020

C++

- Lenguaje multipropósito multiparadigma
- Diferentes estándares:
  - C++98 (o ansi), C++03, **C++11**, C++14, C++17

```
g++ -std=c++11 archivo.cpp
```

- Estándares para C:
  - C90 (o ansi), C99, C11
- Diferentes compiladores: g++, cl
- Portabilidad: el mismo software puede compilar y ejecutar en distintas arquitecturas y compiladores

Tipos Primitivos

- Tipos primitivos y modificadores:
  - bool, char, int, float, double.
  - signed, unsigned, short, long.
  - * (puntero), & (referencia)
  - const (no puede modificar su valor)
- El tamaño de un primitivo es dependientes de la arquitectura. Ej, int puede ser 4 bytes u 8 bytes.
- Para usar tipos con tamaño conocido se deben usar las definiciones de **stdint.h**:
  - int8_t, uint8_t, int32_t, uint64_t
- Otros:
  - size_t se refiere al tamaño de una dirección de memoria
  - wchar_t se refiere al tamaño de un carácter wide

C++

- Fue diseñado para que las clases y sus instancias se puedan comportar igual que un tipo primitivo
  - Overload de operadores
  - Manejo de memoria en el stack (RAII)
- **RAII**: Resource Acquisition Is Initialization
  - Objetos son creados y eliminados en el stack
  - Llamadas implícitas a constructores en la declaración de la variable y a destructores al salir del bloque
    - No hay necesidad de usar **new** o **delete**
  - Duplicación de valores en caso de asignación
  - Duplicación al ser argumentos de una función entregados por valor

Namespaces

- Organización jerárquica de las clases, métodos, constantes.
- El namespace **std** es el utilizado para funciones y objetos de la librería estándar (STL)
  - Ejemplo: **std::string**, **std::map**, **std::cout**, **std::cin**.
- Con la instrucción "`using namespace std`" se puede usar directamente **string**, **map**, **cout**, **cin** y el compilador busca su implementación
  - Usar solo en un .cpp nunca para un header
  - Simplifica el código, pero **no es recomendado** con varios namespaces simultáneamente

Operadores

- Cada clase pueden redefinir los distintos operadores por medio de funciones como la siguientes:
  - `operator+( … )` (suma: a+b)
  - `operator=( … )` (asignación: a=b)
  - `operator[]( … )` (acceso a un elemento: a[0])
  - `operator<<( … )` (shift-left: a << b)
- Cada operador define los parámetros que puede recibir y el tipo de retorno
- Cuando se usa un operador en el código fuente, el compilador determina si existe una implementación definida dependiendo de los tipos de entrada
- En la STL algunas clases implementan los operadores >> y << para los métodos de input/output. Por ej:
  - `std::cout << "hola mundo" << std::endl`

C++

- Un objeto (por ejemplo **std::string**) se comporta como un tipo primitivo (por ejemplo un **int**).
  - El objeto se elimina al terminar su scope
  - La asignación duplica el objeto

```
int c = 1;
int d = c;
d++;
std::cout << c << std::endl;
```

Imprime “1” (d no modifica c)

```
std::string s = "hola";
std::string t = s;  //duplicacion
t.insert(3, "and"); //t es mutable
std::cout << s << std::endl;
```

Imprime “hola” (modificar t no afecta s)

operador de asignación

```
std::vector<int> c;
c.push_back(10);
std::vector<int> d;
d = c;
d.push_back(20);
std::cout << c.size() << std::endl;
```

Imprime “1” (d no modifica c)

Objetos, punteros, referencias

- `Clase c`
  - Declara variable c, crea un objeto tipo Clase (llama al constructor) y lo asigna a c
  - Para leer atributos y llamar métodos del objeto: c.val, c.f()
  - El objeto es destruido cuando *c* sale de su scope (llama al destructor)
  - c siempre contiene un objeto (no puede ser *null)*
- `Clase *p`
  - Declara un puntero (dirección de memoria) donde se encontrará un objeto Clase
  - p se puede dejar sin inicializar o ser *nullptr, NULL*, *0*
  - Para usar el objeto apuntado: p->val, p->f() (si el puntero es inválido ocurre un segfault)
- `Clase &q`
  - Declara una referencia a un objeto tipo Clase
  - q se debe inicializar con un objeto y no se puede modificar (no puede ser *null*)
  - Para usar el objeto apuntado: q.val, q.f()
  - La referencia se elimina al salir del scope (pero no el objeto apuntado)

Operador Puntero y Referencia

- Existen tres operadores `*` `&` `->`
  - No confundir con los tipo de datos! (slide anterior)
  - **`&obj`** “la dirección de obj”
    - Retorna un puntero al objeto obj
  - **`*ptr`** “lo apuntado por ptr”
    - Obtiene el objeto apuntado por ptr
  - **`p->nombre`** atributo de lo apuntado por p
    - Idéntico a usar **`(*p).nombre`**

```
Persona obj; //se crea un objeto
obj.edad = 1;
Persona *p = nullptr; // C++11 (en ANSI se inicializa con 0 o NULL)
p = &obj;
p->edad = 2;
Persona &q = *p;
q.edad = 3;
Persona s; // se crea un nuevo objeto
s = *p; //s copia los atributos de *p
s.edad = 4;
```

Argumentos by-value

- En una función, todos sus argumentos son pasados por valor (copiados)
  - Es lo natural con los tipos primitivos
  - Puede no ser el comportamiento esperado para objetos

Invoca el constructor por defecto, no se usa ()

```
void noagregar(std::vector<int> d) {
  d.push_back(20);
}
int main(int argc, char **argv) {
  std::vector<int> c;
  noagregar(c);
  noagregar(c);
  std::cout << c.size() << std::endl;
}
```

OJO! ***d*** es un duplicado del argumento (se invoca el constructor de copia)

Imprime “0” (no se modifica **c**)

Argumentos by-reference

- En una función, al declarar un argumento como referencia `&` su valor apuntará al objeto mismo que se entregó como parámetro
  - Se puede modificar el argumento
  - Error de compilación si el argumento no es una variable o no se puede modificar

```
void append(std::vector<int> &d) {
  d.push_back(20);
}
int main(int argc, char **argv) {
  std::vector<int> c;
  append(c);
  append(c);
  std::cout << c.size() << std::endl;
}
```

Imprime “2”

Uso de punteros para argumentos

- Es una función, es posible usar punteros como argumentos
  - El puntero mismo se entrega por valor (copiado) pero el objeto apuntado se puede modificar
  - Permite recibir *null* como argumento
  - Enfoque comúnmente usado en Java, pero en C++ se recomienda usar referencias (en lo posible no hacer nunca **new**)

```
void append(std::vector<int> *d) {
  d->push_back(20);
}
int main(int argc, char **argv) {
  std::vector<int> *c = new std::vector<int>();
  append(c);
  append(c);
  std::cout << c->size() << std::endl;
  delete c;
}
```

d (el puntero) es duplicado pero el vector apuntado es el mismo

Imprime “2”

Reglas generales

- Usar by-reference para argumentos “pesados”
  - Cuando sea posible preferir argumentos **`const &`** que aseguran que no se modificará el parámetro (alternativa eficiente a by-value)
  - Argumentos con `&` requieren que se entregue una variable

```
void imprimir(const std::string &str) {
    std::cout << str << std::endl;
}
imprimir("hola");
```

`const&` significa que no se modifica el objeto referenciado

```
void limpiar(std::string &str) {
    str.clear();
}
std::string s = "hola";
limpiar(s);
limpiar(“chao");
```

No compila porque “chao” es una constante

- Usar by-value para argumentos “livianos”
  - Un **int** es más rápido ser copiado que obtener una referencia

Ver Meyers, Item 20

Administración de Memoria

- Es posible pedir memoria con **new** o **new[]** y liberar con **delete** o **delete[]**
- Preferir usar objetos en el stack (RAII) o smart pointers `std::unique_ptr` `std::shared_ptr`

```
std::vector<int> *c = new std::vector<int>();
c->push_back(10);
std::vector<int> *d = c;
d->push_back(20);
std::cout << c->size() << std::endl;
delete c;
```

Imprime “2”

```
std::vector<int> c;
c.push_back(10);
std::vector<int> &d = c;
d.push_back(20);
std::cout << c.size() << std::endl;
```

Imprime “2”

No requiere **delete**!
Imposible que ocurra un memory-leak.

Clases

Área de inicialización de atributos: campo(valor)

Constructor

Destructor (usualmente vacío, ver “regla de tres”)

Este método no modifica el estado interno por lo que puede ser llamado por un “const Complejo”

Método static se invoca Clase::método()

Atributos internos

```
class Complejo {
public:
    Complejo(double real, double imaginario) : a(real), b(imaginario) {
    }
    ~Complejo() {
    }
    void escalar(double s) {
        a *= s;
        b *= s;
    }
    Complejo conjugado() const {
        return Complejo(a, -b);
    }
    void agregar(const Complejo &c) {
        a += c.a;
        b += c.conjugado().b;
    }
    Complejo sumar(const Complejo &c) const {
        return Complejo(a + c.a, b + c.b);
    }

    static Complejo suma(const Complejo &c1, const Complejo &c2) {
        return Complejo(c1.a + c2.a, c1.b + c2.b);
    }
private:
    double a;
    double b;
};
int main(int argc, char **argv) {
    Complejo h(1, 10);
    Complejo i = Complejo::suma(h.conjugado(), Complejo(0, 2));
    i.escalar(2);
    Complejo j = i.sumar(h);
    j.agregar(h.conjugado());
}
```

Templates

- Generación de código durante la compilación en función de parámetros
- Template a nivel de clase y template a nivel de métodos

Template a nivel de clase
T es un tipo por definir

Template a nivel de método
H es un tipo por definir

La clase Complejo<int> tendrá valores a y b de tipo entero

El tipo **H** se define en forma implícita por el tipo del parámetro

```
template<typename T>
class Complejo {
public:
    Complejo(T real, T imag) : a(real), b(imag) {
    }
    template<typename H>
    void escalar(H s) {
        a *= s;
        b *= s;
    }

private:
    T a;
    T b;
};

int main(int argc, char **argv) {
    Complejo<int> n(1, 1);
    Complejo<double> m(1.5, 2.3);
    m.escalar(2);
}
```

Standard Template Library (STL)

- Un template es código fuente generado en tiempo de compilación (distinto a los generics de Java)
  - Se crean clases específicas para cada tipo de dato
- `std::vector<`*T*`>` y `std::deque<`*T*`>`
  - Similar a **ArrayList** y **LinkedList** de Java
  - Operador **vector[*posición*]** permite usarlos como un array
- `std::map<`*K*`,` *T*`>` y `std::unordered_map<`*K*`,` *T*`>`
  - Similar a **TreeMap** y **HashMap** de Java
  - Operador **map[*llave*]** permite usarlos como un arreglo asociativo
- `std::iostream`
  - **std::cout** similar a **System.out** de Java
  - Operadores **<<** y **>>** son re-implementados para hacer I/O
- `std::sstream`
  - **std::ostringstream** similar a **StringBuilder** de Java (notar que en C++ **std::string** es mutable)

Ver Meyers, Item 54

Declaraciones implícitas

- Cuando se declara esto:

```
class Nada {
};
```

el compilador declara e implementa esto

```
class Nada {
public:
 Nada();                                  //constructor por defecto
 Nada(const Nada & otro);                 //constructor de copia
 ~Nada();                                 //destructor
 Nada &operator=(const Nada & otro);      //operador de asignacion
 Nada *operator&();                       //operador de direccion
 const Nada *operator&() const;           //operador de direccion sobre const
};
```

Ver Meyers, Item 5

Orientación a objetos

- Herencia múltiple, herencia privada y pública

Un archivo *.hpp* es un header con declaración de métodos, que luego son implementados en un *.cpp*

Figura.hpp

```
class Figura {
public:
   virtual double getArea()=0;
   virtual ~Figura();
};


class Circulo: public Figura {
public:
   Circulo(double radio);
   virtual double getArea();
private:
   double radio;
};
```

Figura.cpp

```
#include <Figura.hpp>
Figura::~Figura() {
}
Circulo::Circulo(double radio) :
    radio(radio) {
}
double Circulo::getArea() {
   return 3.14 * radio * radio;
}
```

```
Circulo c(1);
std::cout << "A=" << c.getArea() << std::endl;
Figura *f = new Circulo(2);
std::cout << "A=" << f->getArea() << std::endl;
delete f;
```

Orientación a objetos

- Polimorfismo: existen varias implementaciones de un método y en tiempo de ejecución se decide la implementación a ejecutar
  - Al invocar getArea() sobre un objeto tipo Figura se puede ejecutar la implementación de Circulo o Cuadrado dependiendo del objeto
  - En C++ esto se resuelve con una tabla de punteros a funciones
  - **vtable** → Lista de métodos con enlace dinámico
- Los métodos que podrían ser sobre-escritos por una subclase se deben declarar **virtual**
  - El compilador no enlaza directamente al método si no que usa la **vtable**
  - Métodos virtuales sin implementación se declaran con " = 0 " (pure virtual)
- Si existe al menos un método **virtual** entonces el destructor también se debe declarar **virtual**

Ver Meyers, Item 7, 34 y 36

Duplicar Objetos y Destructor

- En C++ es común que los objetos se dupliquen, por ejemplo en paso by-value o en un `std::vector`
- Si un objeto contiene un puntero que inicializa con **new** y el destructor hace **delete**, al duplicar el objeto se copian todos los atributos (incluido el valor del puntero), y los **destructores** de ambos objetos invocarán **delete** sobre el mismo puntero (seg fault!).
  - Solución 1: Usar siempre smart pointers
    - `std::unique_ptr` evita la duplicación del puntero y hace **delete** junto con el objeto.
    - `std::shared_ptr` lleva un contador de referencias, hace un único **delete** cuando nadie lo apunta (ojo con ref-circulares).
  - Solución 2: Usar **new** y no usar **delete** (asumir un memory-leak)
  - Solución 3: Reimplementar el **destructor**, **constructor de copia** y **operador de asignación** (la "regla de tres")

Ver Meyers, Items 13 y 14

Range-Based For-Loop y Auto

- En C++11 se pueden hacer “foreach” y usar variables de tipo “auto”:

```
std::vector<std::string> lista;
for (std::string nombre : lista) { //by-value
    std::cout << nombre << std::endl;
}
for (const std::string &nombre : lista) { //by-const-ref
    std::cout << nombre << std::endl;
}
for (const auto &nombre : lista) { //by-const-ref
    std::cout << nombre << std::endl;
}
```

```
std::map<std::string, std::string> map;
for (std::pair<const std::string, std::string> pair : map) { //by-value
    std::cout << pair.first << "->" << pair.second << std::endl;
}
for (const auto &pair : map) { //by-const-ref
    std::cout << pair.first << "->" << pair.second << std::endl;
}
```

Compilación de código C++

- **Paso 1**: Convertir código fuente en código binario
  - Para cada *.cpp* ejecutar:

```
g++ -c -o archivo1.o archivo1.cpp
```

  - El compilador lee el fuente en el *.cpp* y resuelve todas las instrucciones `#include <librería.hpp>`
  - Con `-Iruta` se configuran lugares para buscar headers además de /usr/include/
  - Agregar `-std=c++11` para usar C++11
  - Agregar `-Wall` para warnings
  - Si el código fuente no tiene problemas se creará un object file .o con la versión binaria del fuente.

Compilación de código C++

- **Paso 2**: Linkeo para crear ejecutable o librería
  - Se reúnen todos los object file y se asocian cada llamada a método con su implementación, ya sea entre object files o con librerías externas (linkage)
  - Para un ejecutable, debe existir un método main() entre todos los *.o*:

```
g++ -o ejemplo archivo1.o archivo2.o
```

  - Para una librería (*.so, .dll*):

```
g++ -shared -o libejemplo.so archivo1.o archivo2.o
```

  - Con **`-Lruta`** se configuran lugares para buscar librerías además de /usr/lib/
  - Con **`-lnombre`** se declaran las librerías externas a buscar (se buscará *libnombre.so* en las rutas definidas).

Compilar código C++

- Es posible compilar y linkear en un solo comando:

```
g++ -o ejemplo -Idir *.cpp -Ldir -llibrerias
```

- El utilitario **pkg-config** entrega los parámetros de compilación al usar librerías externas

```
pkg-config --cflags libreria1 libreria2
pkg-config --libs libreria1 libreria2
```

- Usualmente se escribe un archivo *Makefile* con las instrucciones de compilación que luego se ejecuta con el comando **make**

OPENCV

OpenCV

- Software libre, licencia BSD
- API para lenguajes C, C++, Python, Java
- Versión 4.2.0
- https://opencv.org/
- Compatible con Linux, Android, Windows (mingw/cygwin/visual studio)

OpenCV en C++

- Namespace `cv`
- Tipo `cv::Mat`
  - Objeto con un encabezado con las dimensiones de la imagen y un puntero a los datos de los pixeles
  - Se puede acceder como matriz en la forma (fila,columna) ej:
    - `cv::Mat m;`
    - `m.at<uchar>(y,x)` ← m es 1 canal 8bits (gris)
    - `m.at<cv::Vec3b>(y,x)` ← m es 3 canales 8bits (color)
  - Cuando se copia un `cv::Mat` se duplica el header, pero no los datos
    - Para duplicar la matriz de datos se debe usar .clone()

Reuso de Memoria

- Funciones reciben imagen de entrada y salida

```
threshold(cv::Mat &in, cv::Mat &out, double thresh)
```

  - Si la imagen de salida es vacía se construye su buffer
  - Si la imagen de salida ya existe se reusa el buffer
  - Si la imagen de salida es igual a la de entrada se hace proceso “in-place”
- Documentación (v4.2.0):

https://docs.opencv.org/4.2.0/d2/de8/group__core__array.html

https://docs.opencv.org/4.2.0/d7/d1b/group__imgproc__misc.html

https://docs.opencv.org/4.2.0/d8/dfe/classcv_1_1VideoCapture.html

OpenCV en C++

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
int main(int argc, char** argv) {
    cv::Mat image_color = cv::imread(argv[1], cv::IMREAD_COLOR);
    if (!image_color.data)
        return 0;
    cv::Mat image_gray;
    cv::cvtColor(image_color, image_gray, cv::COLOR_BGR2GRAY);
    cv::Mat image_bin;
    double th = cv::threshold(image_gray, image_bin, 0, 255,
        cv::THRESH_BINARY | cv::THRESH_OTSU);
    std::cout << argv[1] << " threshold: " << th << std::endl;
    cv::imshow("image_color", image_color);
    cv::imshow("image_gray", image_gray);
    cv::imshow("image_bin", image_bin);
    cv::waitKey(0);
    return 0;
}
```

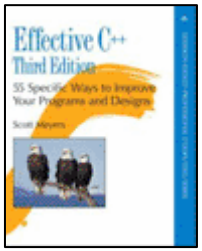
```
g++ -std=gnu++11 -Wall -I[path a includes de opencv]
    -o ejemplo1 ejemplo1.cpp
    -L[path a libs de opencv] -lopencv_core -lopencv_highgui -lopencv_imgproc
```

Ejemplo leyendo un video

```
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>
int main(int argc, char **argv) {
    cv::VideoCapture capture;
    std::string filename = argv[1];
    capture.open(filename);
    if (!capture.isOpened()) {
        int id_webcam = std::atoi(argv[1]);
        capture.open(id_webcam);
    }
    if (!capture.isOpened())
        return 1;
    cv::namedWindow("VIDEO", cv::WINDOW_AUTOSIZE);
    cv::Mat frame, gray;
    for (;;) {
        if (!capture.grab() || !capture.retrieve(frame))
            break;
        cv::cvtColor(frame, gray, cv::COLOR_BGR2GRAY);
        gray = 255 - gray;
        cv::imshow("VIDEO", gray);
        char c = cv::waitKey(33);
        if (c == 27) //tecla ESC
            break;
    }
    return 0;
}
```

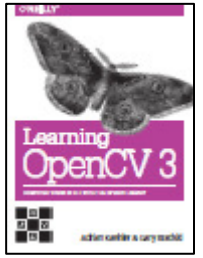
Bibliografía

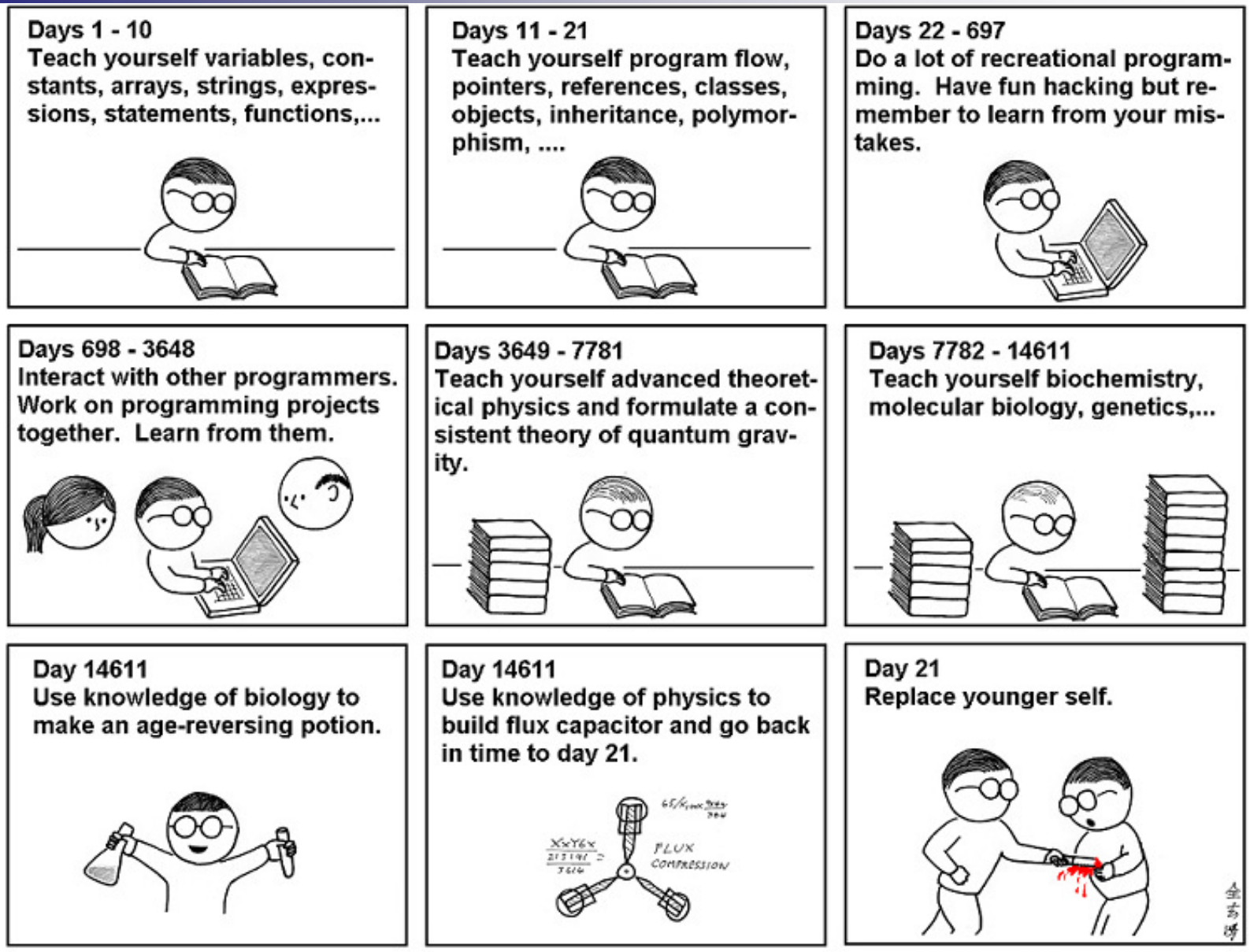
- **Effective C++. Third Edition**. Meyers. 2005.



- **OpenCV 2. Computer Vision Application Programming Cookbook**. Laganière. 2011.

- **Learning OpenCV 3**. Kaehler and Bradski. 2017.





http://abstrusegoose.com/249