ПКШ
Описание и реализация классов

План

1. Различия подходов к представлению данных и процедур.
2. Использование UML для представления объектов и классов.
3. Типы C++.
4. Классы и структуры.
5. Спецификация класса.
6. Специальные члены класса.
7. Управление доступом к классу.
8. Дружественность.
9. Статические члены класса.

К различиям подходов к представлению данных и процедур!

- При **процедурном подходе** вы сначала концентрируетесь на процедурах, которым должен следовать, а только потом думаете о том, как представить данные.
- При **объектно-ориентированном подходе** вы концентрируетесь на объекте, как его представляет пользователь, думая о данных, которые нужны для описания объекта, и операциях, описывающих взаимодействие пользователя с данными.

Абстракции и классы

Абстракции позволяют легко перейти к определяем пользователем типам, которые в C++ представлены классами, реализующий абстрактный интерфейс!

Что такое тип C++

Каждая переменная, аргумент функции и возвращаемое значение функции должны иметь тип, чтобы их можно было скомпилировать.

Вы можете создать **собственный тип**, определив `class` или `struct`.

Тип задает:

- объем памяти, выделяемой для переменной (или результата выражения);
- вид хранимых значений;
- способ интерпретации значений компилятором битовых шаблонов в этих значениях;
- операции, которые можно выполнять со значением.

Скалярный тип

— тип, содержащий одно значение определенного диапазона.

Скаляры включают:

- арифметические типы (целочисленные или значения с плавающей запятой);
- элементы типа перечисления;
- типы указателей;
- типы указателей на члены;
- `std::nullptr_t`.

Основными типами обычно являются скалярные типы.

Составной тип

— тип, который не является скалярным типом 🤓

Составные типы включают типы массивов, типы функций, типы **классов** (или **структур**), типы объединения, перечисления, ссылки и указатели на нестатические члены класса.

Переменная

— символическое имя количества данных. Имя можно использовать для доступа к данным, на которые оно ссылается, в пределах области кода, в которой оно определено.

В C++ переменная часто используется для ссылки на экземпляры скалярных типов данных, тогда как экземпляры других типов обычно называются объектами.

Объект

— экземпляр класса или структуры. При использовании в общем смысле он включает все типы, даже скалярные переменные.

Классы и структуры

Классы и структуры являются конструкциями, в которых пользователь определяет **собственные типы**.

Классы и структуры могут включать данные-члены и функции-члены, позволяющие описывать состояние и поведение данного типа.

Существует три типа классов: структура, класс и объединение.

Они объявляются с помощью ключевых слов `struct`, `class` и `union`. В следующей таблице показаны различия между этими тремя типами классов.

Управление доступом и ограничения для структур, классов и объединений

Структуры	Классы	Объединения
`struct`	`class`	`union`
Доступ по умолчанию: `public` (открытый).	Доступ по умолчанию: `private` (закрытый).	Доступ по умолчанию: `public` (открытый).
Нет ограничений на использование	Нет ограничений на использование	Используется только один член за один раз

Спецификация класса

- **Объявление класса**, описывающее компоненты данных в терминах членов данных, а также открытый интерфейс в терминах функций-членов, называемых **методами**.
- **Определения методов класса**, которое описывают, как реализованы определенные функции-члены.

Грубо говоря:

- **объявление класса** = общий обзор класса;
- **определение методов** класса = детали класса.

Что такое интерфейс?

> **Интерфейс** — это какая-то штука, которая помогает взаимодействовать двум системам или, условно говоря, двум другим штуками. (C) 🤡

Интерфейс – это контракт, который обязуется выполнить класс, выполняющий его, а с другой стороны – тип данных, задающий внешнее поведение объектов, внутреннюю структуру и реализацию которого обеспечивает класс.

P.S. Мы не говорим про то, как интерфейсы реализованы Go, C# или Java. В C++ интерфейсов, строго говоря, нет вообще

Пример объявления класса

```
// TestRun.h
class TestRun {
public:
    TestRun() = default; // Используется конструктор по умолчанию, сгенерированный компилятором:
    TestRun(const TestRun &) = delete; // Не использовать конструктор копирования
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun(); // Деструктор
    enum class State { Active, Suspended }; // Вложенное определение пользовательских типов
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();
private:
    State _state{State::Suspended};
    std::string _testName{""};
    int _index{0};
    static int _instances;
};

int TestRun::_instances{0}; // определить и инициализировать статический элемент.
```

Доступность членов

Содержимое класса обычно находится **в специальных разделах**, разграничивающих **доступ**. Существуют три ключевых слова, которые используются для обозначения начала раздела:

- `private` - закрытый;
- `protected` - защищенный
- `public` - открытый

Данные класса стараются закрыть, спрятать от пользователей класса, а доступ к ним осуществляется через открытые методы.

Обычно используют **заголовочные файлы** `.h`, в которые помещают описание класса. Реализация методов класса обычно размещается в `.cpp`.

Пример объявления класса в `Counter.h`

```
#ifndef _COUNTER_H
#define _COUNTER_H_

typedef unsigned int count_t;

class Counter {
private:
    count_t count;
public:
    void reset();
    void inc();
    count_t get() const;
};
#endif
```

Пример реализации класса в `Counter.cpp`

```cpp
#include "Counter.h"

void Counter::reset() {
    count = 0;
}

void Counter::inc() {
    count++;
}

count_t Counter::get() const {
    return count;
}
```

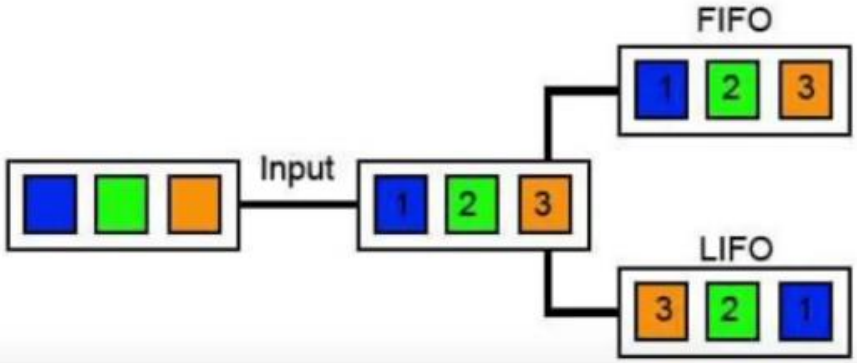
В проекте, где необходимо использовать класс `Counter`, выполняется включение заголовочного файла в те модули, которые используют класс, а далее выполняется компоновка с откомпилированной версией `Counter.cpp`.

Можно использовать следующую команду при работе с `g++` (предполагается, что в `main` основная программа):

```
g++ -o app main.cpp Counter.cpp
```

Вспомним про стек и кучу

- **Стек** — это очень быстрое хранилище памяти, работающее по принципу LIFO и управляемое процессором. Но эти преимущества приводят к ограниченному размеру стека и специальному способу получения значений.
- **Куча** — позволяет создавать динамические и глобальные переменные — но управлять памятью должен либо сборщик мусора, либо сам программист, да и работает куча медленнее.



Создание экземпляров класса (на стеке)

Экземпляры класса (объекты) создаются различными способами. Если мы хотим создать их в **стеке**, подобно обычным автоматическим переменным, то пишем так:

```
int main() {
    Counter pass;
    ...
}
```

Также создаются и массивы объектов:

```
    Counter pass[10];
```

Создание экземпляров класса (на куче)

Самый популярный способ - **создание динамических объектов** (в куче):

```
int main() {
    Counter *pass = new Counter;      // одиночный объект
    Counter *zoo = new Counter[10];   // массив из объектов
    ...
}
```

Не забываем об освобождении динамической памяти:

```
    ...
    delete pass;
    delete[] zoo;
    ...
```

Конструктор - это метод класса, который вызывается при создании объекта, сразу же после выделения объекту памяти.
Основное назначение этого метода - **провести инициализацию полей класса** и **выделить дополнительную память** под внутренние переменные или массивы. Конструкторы бывают:

- **По-умолчанию** - это конструктор без параметров в случае описания и неявный конструктор, назначаемый самой программой в случае отсутствия явного описания.
- **Обычный** - конструктор с параметрами.
- **Копирующий** - создающий копию имеющегося объекта.
- **Перемещающий** - выполняющий действия по перемещению данных их одного объекта в другой (C++11).

Инициализация полей класса непосредственно при объявлении

В стандарте C++11 появилась возможность инициализировать поля класса непосредственно при объявлении:

```
class Counter {
private:
    count_t count = 0;
public:
    ...
};
```

Инициализация полей класса непосредственно при объявлении

При наличии нескольких полей, при инициализации можно ссылаться на значения ранее инициализированных полей:

```
class A {
private:
    count_t count1 = 0;
    count_t count2 = count_1 + 1;
    count_t count3 = count_2 + 1;
public:
    ...
};
```

P.S. Инициализация в конструкторе будет в приоритете.

Пример объявления разных конструкторов

```
class String {
private:
    char *buf; // поле для хранения символьного массива
    size_t len; // длина строки
public:
    String(size_t); // конструктор с параметром числового типа
    String(const char * = nullptr); // конструктор с параметром-указателем
    String(const String &); // конструктор копирования
    String(String &&); // конструктор перемещения
    ...
};
```

Пример реализации разных конструкторов

```
String::String(size_t len) {
    this->len = len;
    buf = new char[len];
    *buf = 0;
}

String::String(const char *str) : String(strlen(str) + 1) {
    strcpy(buf, str);
}

String::String(const String &s) : String(s.len) {
    strcpy(buf, s.buf);
}

String::String(String &&s) {
    buf = s.buf;
    len = s.len;
    s.buf = nullptr;
    s.len = 0;
}
```

Явный вызов конструкторов, то есть операция выделения памяти находится в первом конструкторе и мы вызываем его явно из других:

```
String::String(const char *str) : String(strlen(str) + 1) {
    strcpy(buf, str);
}

String::String(const String &s) : String(s.len) {
    strcpy(buf, s.buf);
}
```

Перемещающий конструктор

Перемещающий конструктор выполняет перенос данных без копирование, то есть захватывает значение указателя, обнуляя оригинал:

```
String::String(String &&s) {
    buf = s.buf;
    len = s.len;
    s.buf = nullptr;
    s.len = 0;
}
```

Деструктор класса - это функция, которая вызывается автоматически при разрушении объекта (или окончания времени жизни). Самое распространенное назначение этого метода - освобождение выделенной в конструкторе динамической памяти:

```
class String {
    ...
public:
    ...
    ~String();
    ...
};

String::~String() {
    delete[] buf;
}
```

Указатель `this`

Указатель `this` позволяет объекту узнать свой адрес или сослаться на члены класса в случае неоднозначности имен-параметров:

```
String(int len) {
    this->len = len; // уточняем имена
}
...
String &get() {
    return *this; // возвращаем ссылку на себя
}
```

Ключевое слово `default`

Еще одной интересной возможностью стандарта C++11 является явное указание того, что тело стандартного метода должно быть выбрано **по-умолчанию**. Для этого существует ключевое слово `default`:

```
class Foo {
public:
    Foo() = default;
    Foo(int x) {/* ... */}
};
```

Ключевое слово `default`

Спецификатор **default** может применяться только к специальным функциям-членам:

- конструктор по-умолчанию;
- конструктор копий;
- конструктор перемещения;
- оператор присваивания;
- оператор перемещения;
- деструктор.

Спецификатор `delete`

Спецификатор `delete` помечает те методы, работать с которыми нельзя. Раньше приходилось объявлять такие конструкторы в приватной области класса.

```
class Foo {
public:
    Foo() = default;
    Foo(const Foo &) = delete;
    void bar(int) = delete;
    void bar(double) {}
};

// ...
Foo obj;
obj.bar(5);     // Call to deleted member function 'bar'
obj.bar(5.42);  // ok
```

Пример использования класса `String`

```
int main() {
    String a("abc");
    String b{"qwerty"};
    String t{move(a)};
    a = move(b);
    b = move(t);
    ...
}
```

`a` создается традиционным способом, а `b` использует **список инициализации** `{}`. В следующих трех строчках реализуется алгоритм обмена содержимым между `a` и `b`. Для задействования конструктора перемещения мы используем функцию `move` из `std`.

Присваивание и инициализация

После того, как переменная была определена, вы можете присвоить ей значение через `=` - **копирующая инициализация** или **присваивание**. Для этого нужны как минимум две инструкции: **определения переменной** и **присвоения значения**. Эти два шага можно совместить.

Переменные можно инициализировать тремя способами:

```
int value1 = 1;      // копирующая инициализация
double value2(2.2); // прямая инициализация
char value3 {'c'};  // унифицированная инициализация
```

Модификатор конструктора `explicit`

Конструктор с таким модификатором `explicit` запрещает создание объекта с преобразованием типа:

```
class String {
    ...
public:
    explicit String(const char *);    // конструктор с параметром числового типа
    ...
};

int main() {
    String a("abc");      // разрешено
    String b{"qwerty"};   // разрешено
    String c = {"123"};   // запрещено! Неявное преобразование строки в объект String
    String d = "zxc";     // запрещено! Неявное преобразование строки в объект String
    ...
```

Модификатор придумали, чтобы **исключить неявные преобразования**.

Инициализация в конструкторе (константы, ссылки)

```
class Mathem {
private:
    const double pi_;
public:
    Mathem(const double pi) {
        pi_ = pi;
    }
};

int main() {
    Mathem m{3.14159};
    return 0;
}
```

При компиляции возникает ошибка:

```
error: constructor for 'Mathem' must explicitly initialize
    the const member 'pi_'
```

Списки инициализаторов членов в конструкторах

```
class Mathem {
private:
    const double pi_;
public:
    Mathem(const double pi) : pi_{pi} {}
};

int main() {
    Mathem m{3.14159};
    return 0;
}
```

Можно и проще...

```
class Mathem {
private:
    const double pi_ = 3.14159;
public:
};

int main() {
    Mathem m;
    return 0;
}
```

Пример с ссылками

```
class A {
};

class B {
private:
    const A &a_;
public:
    B(const A &a) : a_{a} {}
};

int main() {
    A a1;
    B b1{a1};
    return 0;
}
```

Управление доступом к классу (инкапсуляция)

Главная забота класса - **скрыть как можно больше информации**. Существует 4 вида пользователей класса:

1. сам класс;
2. обычные пользователи (другие классы);
3. производные классы;
4. дружественные классы;

Каждый пользователь обладает привилегиями доступа к членам класса.

Правила доступа к элементам класса

- Сам **класс** имеет полный доступ ко всем своим элементам;
- Обычные пользователи (**другие классы**) имеют полный доступ только к открытому (`public`) разделу;
- **Производные классы** имеют доступ к `public`-разделу и к разделу `protected`;
- **Дружественные классы** и **функции** имеют полный доступ ко всем разделам.

Разумеется, для внешних пользователей класса, приватные члены доступны через интерфейсные `public`-методы.

Дружественность

Дружественный класс имеет доступ как к общедоступным, так и к закрытым членам другого класса. С ключевым словом **friend** может быть объявлен как целый класс, так и отдельная функция.

```
class MyClass1 {
    friend MyClass2;
    ...
};

class MyClass2 {
    ...
};
```

Отношение дружественности не наследуется!

Полное и неполное объявление класса

Неполное объявление используется для ссылки на класс, который еще не совсем определен, например, когда он находится в **другом файле** или **расположен ниже** по тексту в этом же файле.

```
class MyClass; // ПРЕДВАРИТЕЛЬНОЕ ОБЪЯВЛЕНИЕ

MyClass *mc; // для этого мы добавили неполное объявление

class MyClass { // ПОЛНОЕ ОПИСАНИЕ
    ......
};
```

Статические члены класса

В классе мы можем использовать поля и методы с ключевым словом `static`. Особенность их в том, что они принадлежат именно классам, а не конкретным экземплярам, то есть не тиражируются при создании объектов.

Статические переменные **должны инициализироваться вне классов**, вне зависимости от уровня доступа.

Пример использования статических членов класса

```cpp
#include <iostream>

class Something {
private:
    static int s_value;
public:
    static int get() {
        return s_value;
    }
};

int Something::s_value{1};

int main() {
    std::cout << Something::get() << '\n';
    return 0;
}
```

Популярный пример использования статических членов класса

Один из популярных примеров использования статических переменных - **счетчики объектов**.

В конструкторе данного класса размещается операция **инкремента** счетчика, а в деструкторе - **декремента**. Запросив значение счетчика мы можем узнать, какое количество объектов существует на данный момент.

В C++ появились новые возможности

1. поддержка объектно-ориентированного программирования
2. поддержка обобщённого программирования через шаблоны
3. дополнительные типы данных
4. исключения
5. пространства имён
6. встраиваемые функции
7. перегрузка операторов
8. перегрузка имён функций
9. ссылки и операторы управления свободно распределяемой памятью
10. дополнения к стандартной библиотеке

Переход с **Си** на **C++**

По замечанию Страуструпа, «чем лучше вы знаете C, тем труднее вам будет избежать программирования на C++ в стиле C, теряя при этом потенциальные преимущества C++». В связи с этим он даёт следующий набор рекомендаций для программистов на C, чтобы в полной мере воспользоваться преимуществами C++:

1. Не использовать макроопределения #define. Для объявления констант применять const, групп констант (перечислений) — enum, для прямого включения функций — inline, для определения семейств функций или типов — template.

Переход с **Си** на **C++**

2. Не использовать предварительные объявления переменных. Объявлять переменные в блоке, где они реально используются, всегда совмещая объявление с инициализацией.

3. Отказаться от использования malloc() в пользу оператора new, от realloc() — в пользу типа vector.

4. Избегать бестиповых указателей, арифметики указателей, неявных приведений типов.

5. Свести к минимуму использование массивов символов и строк в стиле C, заменив их на типы string и vector из STL.

Почему используют Си?

За счет ручного управления памятью. Как отмечается в исследованиях, программисты на Си тратят 30 % — 40 % общего времени разработки только на управление памятью. Считается, что Си разработан для решения низкоуровневых задачах и там это оправданно,

C++ разработан для решения прикладных задачах широкого спектра, там ручное управления памятью является не только напрасным, но и чревато ошибками.

Style

Consistency is the most important aspect of style. The second most important aspect is following a style that the average C++ programmer is used to reading.

C++ allows for arbitrary-length identifier names, so there's no reason to be terse when naming things. Use descriptive names, and be consistent in the style.

- `CamelCase`
- `snake_case`

are common examples. *snake_case* has the advantage that it can also work with spell checkers, if desired.

style you expect. While this cannot help with naming, it is particularly important for an open source project to maintain a consistent style.

Every IDE and many editors have support for clang-format built in or easily installable with an add-in.

- VSCode: Microsoft C/C++ extension for VS Code
- CLion: https://www.jetbrains.com/help/clion/clangformat-as-alternative-formatter.html
- VisualStudio https://marketplace.visualstudio.com/items?itemName=LLVMExtensions.ClangFormat#review-details
- Resharper++: https://www.jetbrains.com/help/resharper/2017.2/Using_Clang_Format.html
- Vim
  - https://github.com/rhysd/vim-clang-format
  - https://github.com/chiel92/vim-autoformat
- XCode: https://github.com/travisjeffery/ClangFormat-Xcode

Common C++ Naming Conventions

- Types start with upper case: `MyClass`.
- Functions and variables start with lower case: `myMethod`.
- Constants are all upper case: `const double PI=3.14159265358979323;`.

C++ Standard Library (and other well-known C++ libraries like Boost) use these guidelines:

- Macro names use upper case with underscores: `INT_MAX`.
- Template parameter names use Pascal case: `InputIterator`.
- All other names use snake case: `unordered_map`.

Distinguish Private Object Data

Name private data with a `m_` prefix to distinguish it from public data. `m_` stands for "member" data.

Distinguish Function Parameters

The most important thing is consistency within your codebase; this is one possibility to help with consistency.

Name function parameters with an `t_` prefix. `t_` can be thought of as "the", but the meaning is arbitrary. The point is to distinguish function parameters from other variables in scope while giving us a consistent naming strategy.

Any prefix or postfix can be chosen for your organization. This is just one example. *This suggestion is controversial, for a discussion about it see issue #11.*

```
struct Size
{
  int width;
  int height;

  Size(int t_width, int t_height) : width(t_width), height(t_height) {}
};

// This version might make sense for thread safety or something,
// but more to the point, sometimes we need to hide data, sometimes we don't.
class PrivateSize
{
  public:
    int width() const { return m_width; }
```

Don't Name Anything Starting With _

If you do, you risk colliding with names reserved for compiler and standard library implementation use:

http://stackoverflow.com/questions/228783/what-are-the-rules-about-using-an-underscore-in-a-c-identifier

```cpp
class MyClass
{
public:
  MyClass(int t_data)
    : m_data(t_data)
  {
  }

  int getData() const
  {
    return m_data;
  }

private:
  int m_data;
};
```

Use `nullptr`

C++11 introduces `nullptr` which is a special value denoting a null pointer. This should be used instead of `0` or `NULL` to indicate a null pointer.

Comments

Comment blocks should use `//`, not `/* */`. Using `//` makes it much easier to comment out a block of code while debugging.

Never Use `using namespace` in a Header File

This causes the namespace you are `using` to be pulled into the namespace of all files that include the header file. It pollutes the namespace and it may lead to name collisions in the future. Writing `using namespace` in an implementation file is fine though.

Include Guards

Header files must contain a distinctly-named include guard to avoid problems with including the same header multiple times and to prevent conflicts with headers from other projects.

```
#ifndef MYPROJECT_MYCLASS_HPP
#define MYPROJECT_MYCLASS_HPP

namespace MyProject {
  class MyClass {
  };
}

#endif
```

{} Are Required for Blocks.

Leaving them off can lead to semantic errors in the code.

```
// Bad Idea
// This compiles and does what you want, but can lead to confusing
// errors if modification are made in the future and close attention
// is not paid.
for (int i = 0; i < 15; ++i)
  std::cout << i << std::endl;
```

```
// Good Idea
// It's clear which statements are part of the loop (or if block, or whatever).
int sum = 0;
for (int i = 0; i < 15; ++i) {
  ++sum;
  std::cout << i << std::endl;
}
```

Keep Lines a Reasonable Length

```
// Bad Idea
// hard to follow
if (x && y && myFunctionThatReturnsBool() && caseNumber3 && (15 > 12 || 2 < 3)) {
}

// Good Idea
// Logical grouping, easier to read
if (x && y && myFunctionThatReturnsBool()
    && caseNumber3
    && (15 > 12 || 2 < 3)) {
}
```

Use "" for Including Local Files

... `<>` is reserved for system includes.

```
// Bad Idea. Requires extra -I directives to the compiler
// and goes against standards.
#include <string>
#include <includes/MyHeader.hpp>

// Worse Idea
// Requires potentially even more specific -I directives and
// makes code more difficult to package and distribute.
#include <string>
#include <MyHeader.hpp>
```

Initialize Member Variables

...with the member initializer list.

For POD types, the performance of an initializer list is the same as manual initialization, but for other types there is a clear performance gain, see below.

```
// Bad Idea
class MyClass
{
public:
  MyClass(int t_value)
  {
    m_value = t_value;
  }
```

```
// Bad Idea
// This leads to an additional constructor call for m_myOtherClass
// before the assignment.
class MyClass
{
public:
  MyClass(MyOtherClass t_myOtherClass)
  {
    m_myOtherClass = t_myOtherClass;
  }

private:
  MyOtherClass m_myOtherClass;
};
```

```
// Good Idea
// The default constructor for m_myOtherClass is never called here, so
// there is a performance gain if MyOtherClass is not is_trivially_default_constructib
class MyClass
{
public:
  MyClass(MyOtherClass t_myOtherClass)
    : m_myOtherClass(t_myOtherClass)
  {
  }

private:
  MyOtherClass m_myOtherClass;
};
```

In C++11 you can assign default values to each member (using `=` or using `{}`).

Assigning default values with =

```
// ... //
private:
  int m_value = 0; // allowed
  unsigned m_value_2 = -1; // narrowing from signed to unsigned allowed
// ... //
```

This ensures that no constructor ever "forgets" to initialize a member object.

Always Use Namespaces

There is almost never a reason to declare an identifier in the global namespace. Instead, functions and classes should exist in an appropriately named namespace or in a class inside of a namespace. Identifiers which are placed in the global namespace risk conflicting with identifiers from other libraries (mostly C, which doesn't have namespaces).

Use .hpp and .cpp for Your File Extensions

Ultimately this is a matter of preference, but .hpp and .cpp are widely recognized by various editors and tools. So the choice is pragmatic. Specifically, Visual Studio only automatically recognizes .cpp and .cxx for C++ files, and Vim doesn't necessarily recognize .cc as a C++ file.

One particularly large project (OpenStudio) uses .hpp and .cpp for user-generated files and .hxx and .cxx for tool-generated files. Both are well recognized and having the distinction is helpful.

Don't Be Afraid of Templates

They can help you stick to DRY principles. They should be preferred to macros, because macros do not honor namespaces, etc.